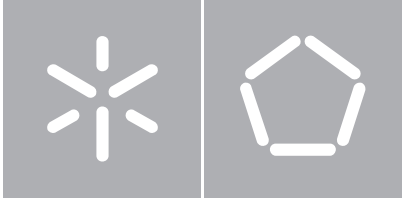
Universidade do Minho
Escola de Engenharia

André Couto da Silva

Conceção e desenvolvimento de uma aplicação Windows 8 numa arquitetura orientada a serviços.

Outubro de 2013



Universidade do Minho
Escola de Engenharia
Departamento de Informática

André Couto da Silva

Conceção e desenvolvimento de uma aplicação Windows 8 numa arquitetura orientada a serviços.

Dissertação de Mestrado
Mestrado em Engenharia Informática

Trabalho realizado sob orientação de

Professor Doutor António Manuel Nestor Ribeiro

Outubro de 2013

Agradecimentos

A conclusão desta dissertação marca o fim de uma etapa da minha vida, talvez uma das mais importantes.

Como tal, gostaria de começar por agradecer aos meus pais, por tudo o que me deram, em particular a educação e a possibilidade de chegar a este ponto no meu percurso académico. À Joana, por me ter apoiado sempre que precisei e por me relembrar, insistentemente, dos objetivos a que me propus. Foi graças a ela que tive o equilíbrio necessário para que o trabalho corresse sem percalços.

Ao Professor Nestor Ribeiro pelas orientações que me deu e por me ter ajudado a encontrar o rumo certo na abordagem ao problema. De igual modo deixo o meu agradecimento à Primavera BSS e em particular ao Lourenço Antunes, por ter acreditado no meu valor e me ter dado a oportunidade de desenvolver este trabalho com toda a tranquilidade e apoio que precisei. Aos meus colegas de trabalho por todo o conhecimento que partilharam comigo, porque sem eles seria impossível concluir esta dissertação. Em particular gostaria de deixar uma palavra especial de agradecimento ao David Resende, por todo o tempo perdido e enorme conhecimento partilhado, ao Miguel Pinto por todos os conselhos, preocupação e acompanhamento, ao Miguel Marques pelas sempre pertinentes chamadas de atenção, ao Luís Quental, pelas ideias fora da caixa e a todos os outros que de uma ou de outra forma foram dando algum contributo no dia-a-dia.

Obrigado.

André Couto da Silva

Outubro de 2013

Título

Conceção e desenvolvimento de uma aplicação Windows 8 numa arquitetura orientada a serviços.

Resumo

A troca de informação acontece hoje muito rapidamente, sendo possível efetuar tarefas colaborativas mais facilmente, o que permite atingir uma maior eficiência e desempenho. Os serviços *web* são uma tecnologia que tem emergido neste âmbito, permitindo a implementação mais fácil de interações entre aplicações heterogéneas. Foram desenvolvidos vários padrões como o WSDL e o SOAP para suportar o desenvolvimento de serviços *web,* mas, ao mesmo tempo, têm surgido outras formas de serviços que recorrem aos princípios arquiteturais REST e ao protocolo http. Nesta dissertação foi feito um estudo, no contexto da empresa Primavera BSS, do desenvolvimento de *software* usando os princípios de uma arquitetura orientada a serviços. Foi concebida uma solução entregue como um serviço utilizando a infraestrutura Primavera CloudServices, que visa o desenvolvimento deste tipo de soluções e, ainda, uma aplicação da loja de aplicações Windows da Microsoft. O objetivo da conceção desta aplicação foi o de perceber como estas podem ser usadas no contexto de uma arquitetura orientada a serviços e quais as mudanças face às tradicionais aplicações para sistemas operativos Microsoft Windows. Concluiu-se que o desenvolvimento de *software* usando este tipo de arquitetura abre espaço para novas funcionalidades e providencia uma forma de interoperabilidade com sistemas mais antigos. Além disso, percebeu-se que as aplicações da loja Microsoft para sistemas Windows 8/RT não são um substituto de aplicações mais tradicionais, mas podem complementar as mesmas.

Palavras-chave: Arquitetura Orientada a Serviços; Serviços *Web*; Aplicação Windows; SOAP; REST; Interoperabilidade.

Title

Design and development of a Windows 8 application on a service-oriented architecture.

Abstract

Information exchange happens nowadays very quickly, and it is possible to perform collaborative tasks more easily, which allows for the achieving of higher efficiency and performance. Web services are a technology that has emerged, allowing the easier implementation of interactions between heterogeneous applications. Several standards were developed to support the development of web services, such as WSDL and SOAP, but at the same time, other forms of services, based on the REST architectural principles and the http protocol, have emerged. In this dissertation, in the context of the company Primavera BSS, a study of software development using the principles of service oriented architecture has been done. A solution delivered as a service was designed and developed using the Primavera CloudServices infrastructure that aims at the development of such solutions. A Microsoft Windows Store Application was also developed. The rationale behind its conception was to understand how can this kind of applications be used in the context of a service oriented architecture, and which are the changes comparing it with traditional applications of Microsoft's operating systems. We conclude that using service oriented architecture in software development makes room for new functionality and provides a form of interoperability with legacy systems. In addition, it was concluded that Microsoft Windows Store Applications are not a replacement for more traditional applications but can complement them.

Keywords: Service oriented architecture; Web Services; Windows Applications; SOAP; REST; Interoperability.

Conteúdo

AGRADECIMENTOS ..... **III**

RESUMO ..... **V**

ABSTRACT ..... **VII**

CONTEÚDO ..... **IX**

ÍNDICE DE FIGURAS ..... **XI**

LISTAGENS ..... **XIII**

LISTA DE SIGLAS E ACRÓNIMOS ..... **XV**

CAPÍTULO 1 INTRODUÇÃO **1**

1.1 OBJETIVOS 3

1.2 ESTRUTURA DO DOCUMENTO 4

CAPÍTULO 2 ESTADO DA ARTE **7**

2.1 ARQUITETURA ORIENTADA A SERVIÇOS 8

2.1.1 Modelo de interação 12

2.2 SERVIÇOS WEB 15

2.2.1 Protocolos 16

2.2.2 Arquiteturas de Serviços Web 21

2.3 MODELOS DE IMPLEMENTAÇÃO DE ERP E A COMPUTAÇÃO NA NUVEM 28

2.3.1 Computação na Nuvem 29

2.3.2 ERP híbridos 31

2.4 APLICAÇÕES DA LOJA WINDOWS 32

2.4.1 Experiência de Utilização 33

CAPÍTULO 3 ANÁLISE DE REQUISITOS DO CASO DE ESTUDO **35**

3.1 Cenário 35
3.2 Storyboard 36
3.3 Modelo de Domínio 39
3.4 User Stories 40
3.5 Casos de Uso 43
3.6 Conclusões da Análise de Requisitos 45
3.6.1 Notes 45
CAPÍTULO 4 ARQUITETURA 47
4.1 Infraestrutura 47
4.1.1 Framework Athena 47
4.1.2 Primavera Elevation 52
4.2 Arquitetura Implementada 54
4.2.1 Arquitetura do Módulo NotesAthena 55
4.2.2 Arquitetura do Módulo NotesWebCentral 55
4.2.3 Aplicações Cliente 57
CAPÍTULO 5 DESENVOLVIMENTO DO CASO DE ESTUDO 61
5.1 Implementação do Módulo NotesAthena 61
5.1.1 Entidades NotesAthena 62
5.1.2 Serviços NotesAthena 63
5.2 Implementação do Módulo NotesWebCentral 67
5.2.1 Objetos de Transferência de Dados NotesWebCentral 67
5.2.2 Serviços NotesWebCentral 68
5.3 Aplicações Cliente 71
5.3.1 Windows Store App 71
5.3.2 Controlo Web NotesWebCentral 77
5.3.3 Interação com o utilizador 81
CAPÍTULO 6 CONCLUSÕES 87
6.1 Trabalho futuro 90
GLOSSÁRIO 91
BIBLIOGRAFIA 92

Índice de Figuras

Figura 2.1 – Evolução das arquiteturas 7
Figura 2.2 – Camadas de uma Arquitetura SOA 11
Figura 2.3 – Serviços 12
Figura 2.4 – Modelo de Interação 13
Figura 3.1 – *Storyboard* 38
Figura 3.2 – Modelo de Domínio. 39
Figura 3.3 – Diagrama de casos de uso 43
Figura 4.1 – Arquitetura de uma solução Athena 48
Figura 4.2 – Arquitetura de um ambiente multinível 49
Figura 4.3 – Estrutura de um produto Athena 50
Figura 4.4 – Diagrama da framework de modelação 51
Figura 4.5 – Arquitetura Soluções Elevation 53
Figura 4.6 – Arquitetura da infraestrutura dos serviços desenvolvida 54
Figura 4.7 – Diagrama de uma arquitetura MVVM 58
Figura 5.1 – Modelo de Entidades NotesAthena 62
Figura 5.2 – Modelo de serviços NotesAthena 65
Figura 5.3 – Diagrama de Classes dos DTO *(DataTransferObjects)* 68
Figura 5.4 – Diagrama da arquitetura da aplicação Windows 74
Figura 5.5 – Captura de ecrã "Criar Tópico" 81
Figura 5.6 – Captura de ecrã "Partilhar Tópico" 81
Figura 5.7 – Captura de ecrã "Consultar Tópicos Partilhados" 82
Figura 5.8 – Captura de ecrã "Consultar Tópicos Partilhados - UserSpace" 83
Figura 5.9 – Captura de ecrã "Consultar Tópicos Partilhados – ERP" 83
Figura 5.10 – Captura de ecrã "Responder a Tópico" 84

Figura 5.11 – Captura de ecrã "Consultar Notas Relacionadas" 84
Figura 5.12 – Captura de ecrã "Pesquisar Notas" 85
Figura 5.13 – Captura de ecrã "Filtrar Notas" 85

Listagens

Listagem 2.1 – Um pedido http GET enviado ao *host* http://primaverabss.com 16
Listagem 2.2 – Resposta http a um pedido GET 17
Listagem 2.3 – Mensagem SOAP 19
Listagem 2.4 – Fração de um documento WSDL 20
Listagem 2.5 – Pedido fictício ao recurso /clientes 22
Listagem 2.6 – Resposta ao pedido 22
Listagem 2.7 – Documento XML de um pedido RPC 24
Listagem 2.8 – Exemplo de uma interface C# que expõe uma operação 25
Listagem 3.1 – Caso de uso: "Responder a um tópico" 44
Listagem 5.1 – Método C# GetNotesByUserId da classe NoteManager 66
Listagem 5.2 – API de serviços NotesWebCentral 69
Listagem 5.3 – Conteúdo de um POST ao serviço /notes 70
Listagem 5.4 – Excerto do método C# PostNote da classe NoteManager 71
Listagem 5.5 – Alternativas de implementação das Windows Store Apps 72
Listagem 5.6 – Exemplo de data-binding 74
Listagem 5.7 – Excerto de código da *DashboardView.xaml* 75
Listagem 5.8 – Método PostNote do DashboardViewModel 76
Listagem 5.9 – Método PostNote da classe *ClientDataService* 77
Listagem 5.10 – Função PostNote do controlo NotesWebCentral 78
Listagem 5.11 – Exemplo da estrutura XML que guarda um contexto 80

Lista de Siglas e Acrónimos

API	Application Programming Interface
CRUD	Create, Read, Update, Delete
ERP	Enterprise Resource Planning
HTML	HyperText Markup Language
HTTP	HyperText Transfer Protocol
HTTPS	HyperText Transfer Protocol Secure
MVVM	Model – View - ViewModel
OASIS	Organization for the Advancement of Structured Information Standards
ORM	Object-Relational Mapping
OSI	Open Systems Interconnection
REST	Representational State Transfer
SDK	Software Development Kit
SOA	Service Oriented Architecture
SSL	Secure Sockets Layer
SWOT	Strengths, Weaknesses, Opportunities, Threats
TIC	Tecnologias da Informação e Comunicação
TLS	Transport Layer Security
URI	Uniform Resource Identifier
UX	User Experience
WCF	Windows Comunication Foundation
WSDL	Web Services Description Language
XAML	eXtensible Application Markup Language
XML	eXtensible Markup Language

Capítulo 1

Introdução

Um dos problemas que as empresas da área das *Tecnologias da Informação e Comunicação* (TIC) têm enfrentado nos últimos anos é o de perspetivarem em que tecnologias devem apostar para desenvolver o seu *software*. Existe uma alargada oferta de soluções suportada por diferentes paradigmas, plataformas e ferramentas de desenvolvimento. Esta oferta é quase sempre propriedade de uma determinada empresa, responsável pela sua manutenção e evolução. Desta forma, a escolha de um determinado ambiente de desenvolvimento pode ser uma decisão crítica para uma empresa de TIC tendo em conta os riscos da aposta numa determinada tecnologia. Este facto provoca alguma pressão sobre as empresas que queiram investir os seus recursos em algo que lhes possa trazer retorno, não só a curto, como a médio/longo prazo.

Uma situação que se verifica em algumas empresas com alguns anos no setor é a existência de *software* legado *(legacy software).* A área das ciências da computação pode ser recente quando comparada com outras áreas do conhecimento científico, contudo, tem idade suficiente para já existir muito *software* considerado obsoleto pelos padrões atuais. Muitas empresas têm em sua posse muitos anos de desenvolvimento e conhecimento de negócio nesses sistemas, para além de um alargado número de clientes dependente dos mesmos, tornando o custo de modernização desses sistemas simplesmente incomportável para ser feito em tempo útil e de forma rentável. A par desta situação verifica-se, por parte das empresas, o medo de perder dados ou falhar a transição para um novo paradigma.

A realidade atual da área das TIC é a existência de um mundo composto por sistemas muito heterogéneos, cada vez mais globalizado, e em que as empresas querem sempre ganhar vantagem competitiva sobre a concorrência. A exigência é cada vez maior e obriga as empresas a adaptarem-se a uma nova realidade.

Um dos desafios nesta área é encontrar uma forma de desenvolver *software* que permita ultrapassar ou atenuar estes problemas para que, por um lado, se evite que se possam perder anos e anos de desenvolvimento de software e, por outro lado, se potencie o desenvolvimento de sistemas que permitam evitar que, no futuro, voltem a acontecer os mesmos problemas do passado. Neste sentido, surgiu o interesse na presente dissertação, a qual foi realizada no âmbito de um projeto a desenvolver na empresa *Primavera Business Software Solutions* (Primavera BSS).

A Primavera BSS é uma empresa de *software,* que se dedica desde 1993 ao desenvolvimento e comercialização de soluções de gestão e plataformas para integração de processos empresariais num mercado global, disponibilizando soluções para as Pequenas, Médias, Grandes Organizações e Administração Pública. Os principais produtos desenvolvidos tradicionalmente por esta empresa consistem em soluções direcionadas a ambientes Microsoft Windows®, construídas com recurso a tecnologia maioritariamente Microsoft, que são instalados localmente nestas organizações. O principal produto da Primavera é o ERP *(Enterprise Resource Planning)* Primavera, que permite integrar todos os dados e processos de uma organização num único sistema, facilitando a automatização de processos.

Com esta dissertação pretende-se conceber uma solução entregue como um serviço[1] e investigar de que forma este serviço pode integrar com as soluções locais tradicionalmente desenvolvidas pela Primavera. O serviço a desenvolver será um serviço de partilha de anotações sobre documentos criados no ERP Primavera, com o principal objetivo de facilitar a comunicação entre as pessoas intervenientes nos processos tratados pelo ERP. Pretende-se ainda estudar se a integração deste tipo de serviços com os sistemas locais pode abrir espaço para o desenvolvimento de funcionalidades sustentáveis, sob as quais seja mais fácil comunicar com outro tipo de plataformas e sem que seja necessário destruir o *software* legado já existente, ou até encontrar uma nova utilidade para o mesmo. A Primavera tem, aliás, em desenvolvimento, uma *framework*, denominada Athena que se baseia nos princípios do desenvolvimento de *Software* Orientado a Modelos *(Model-Driven Development)* e cujo objetivo principal é servir o desenvolvimento dos produtos Primavera. Um dos projetos desenvolvidos com recurso a esta *framework* é o projeto Primavera *Cloud Services,* que servirá de infraestrutura para o

[1] Vulgarmente conhecido como serviço web ou *web-service*, no sentido mais lato do termo e não necessariamente na implementação concreta da chamada tecnologia de *web-services.*

desenvolvimento do serviço que é âmbito desta dissertação. Este projeto está incluído num conjunto de soluções, agrupado sob o nome Primavera Elevation, cujo ponto de contacto entre todos é a sua ligação com ambientes *web/cloud.*

Saliente-se que é também objetivo deste trabalho estudar a forma como as pessoas podem tirar partido deste tipo de soluções orientadas a serviços, no seu âmbito pessoal ou doméstico, com a conceção de uma aplicação para uso pessoal, mais focada nas pessoas e menos focada nos processos das organizações, que tem sido até agora o foco dos sistemas desenvolvidos pela Primavera. Esta será uma aplicação da loja de aplicações da Microsoft para sistemas Microsoft Windows 8/RT e cujo objetivo de conceção é ajudar a perceber qual a sustentabilidade deste tipo de aplicações no contexto de uma arquitetura orientada a serviços.

1.1 Objetivos

Tal como foi enunciado na introdução, com esta dissertação pretende-se conceber uma solução orientada a serviços que permita realizar uma análise a este tipo de desenvolvimento de *software*. A construção do serviço será feita usando como infraestrutura o projeto Primavera *Cloud Services*, que recorre à já referida *Framework Athena*. A construção do caso prático deve incluir também o desenvolvimento de uma aplicação da loja de aplicações da Microsoft para sistemas Microsoft Windows 8/RT que também faça uso do serviço concebido.

Podem assim definir-se os seguintes objetivos principais:

- Elaborar um estudo da arquitetura orientada a serviços e às diferentes formas de implementação de uma arquitetura desse estilo. Para isso será realizada uma breve análise a este tipo de arquitetura e posteriormente uma comparação entre diferentes formas de implementação de serviços *web*.
- Investigar o estado atual do desenvolvimento de ERP's e tentar perceber de que forma pode uma arquitetura orientada a serviços ser usada na construção destes ou em soluções paralelas que potenciem os sistemas atuais.
- Estudar as aplicações da loja de aplicações da Microsoft para Windows 8/RT e perceber como podem ser usadas num cenário de uma arquitetura orientada a serviços e quais as mudanças face às tradicionais aplicações para sistemas operativos Microsoft Windows.

- Analisar o cenário de partilha de anotações sobre documentos ou entidades criadas no ERP e fazer um levantamento de requisitos que um serviço deste tipo deve cumprir. Deste levantamento de requisitos devem ficar definidas as funcionalidades a implementar, tanto no serviço como na aplicação Windows 8/RT.
- Conceber um serviço utilizando a infraestrutura Primavera *Cloud Services*, implementando as funcionalidades levantadas na análise de requisitos.
- Conceber uma aplicação da loja de aplicações da Microsoft para sistemas Microsoft Windows 8/RT.

É esperado que na conclusão desta dissertação possa ser feita uma análise sobre o padrão de desenvolvimento de funcionalidades disponibilizadas na forma de serviços, ou seja, usando uma arquitetura orientada a serviços. Pretende-se também fazer uma avaliação das tecnologias utilizadas, nomeadamente da tecnologia Primavera usada como suporte, assim como do desenvolvimento de aplicações da loja de aplicações da Microsoft.

1.2 Estrutura do Documento

Esta dissertação seguirá a estrutura que resumidamente aqui se apresenta:

- **Capítulo 2:** Neste capítulo será feita uma investigação sobre as temáticas em análise nesta dissertação. O segundo capítulo é dividido em quatro partes. Numa primeira parte será feito um estudo sobre a arquitetura orientada a serviços, a qual será seguida de uma análise às diferentes formas de implementar uma arquitetura deste tipo. Seguir-se-á uma análise à perspetiva do desenvolvimento de soluções empresariais que integrem com o paradigma da *Cloud.* Na quarta e última parte serão apresentadas as características das aplicações da loja de aplicações da Microsoft.
- **Capítulo 3:** No terceiro capítulo será feita uma análise de requisitos do caso de estudo. Será realizada uma análise ao cenário de partilha de anotações sobre documentos ou entidades criadas no ERP.
- **Capítulo 4:** Este capítulo será dividido em duas partes. Na primeira será feita uma explicação das tecnologias Primavera usadas na conceção do projeto. Na segunda será descrito o desenho da implementação do projeto. Serão apresentados os diagramas de arquitetura, sendo cada componente desenvolvido descrito em detalhe.

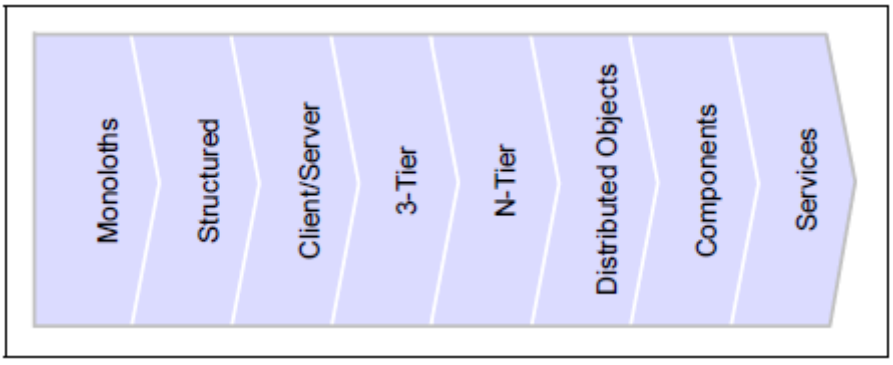
- **Capítulo 5:** No quinto capítulo serão detalhadas as decisões de implementação e demonstradas capturas de ecrã da aplicação e controlo desenvolvidos.
- **Capítulo 6:** No sexto capítulo serão retiradas as conclusões sobre o desenvolvimento de funcionalidades disponibilizadas na forma de serviços, e sobre o desenvolvimento de aplicações da loja de aplicações da Microsoft. É também no sexto capítulo que são lançadas algumas ideias para trabalho futuro.

Capítulo 2

Estado da Arte

O interesse pela resolução dos problemas enunciados na introdução deste documento não é recente e a arquitetura dos sistemas de *software* tem sofrido uma evolução ao longo dos tempos. A **Figura 2.1** demonstra a evolução destas arquiteturas, inicialmente maioritariamente monolíticas, em que o código das aplicações de *software* era combinado num único programa, responsável por todas as tarefas. Este tipo de arquitetura de baixa/nenhuma modularidade, tem níveis de reutilização baixos e como tal, foi evoluindo para outras arquiteturas mais modulares como as arquiteturas de várias camadas, em que cada uma destas é responsável por uma tarefa (e.g.: camada de acesso e manipulação de dados, camada das regras de negócio, etc.). Mais recentemente o mundo evoluiu para um ecossistema muito mais heterogéneo, com aplicações concebidas para ambientes completamente distintos, mas que agora se espera que comuniquem com o restante ecossistema.

Figura 2.1 – Evolução das arquiteturas



Fonte: Endrei et al. (2004)

Para aliviar os problemas de heterogeneidade, interoperabilidade e de requisitos em constante mudança, uma arquitetura deve providenciar uma plataforma para construir serviços com características de baixa dependência, localização transparente e independência de

protocolo de comunicação (Endrei et al., 2004). Uma arquitetura que permite cumprir essas características é a arquitetura orientada a serviços, onde a infraestrutura esconde o mais possível a tecnologia em que os serviços assentam, permitindo, assim, a integração de diferentes tecnologias, as quais normalmente teriam dificuldades em ser integradas. Por outro lado existem ainda algumas dúvidas acerca da facilidade com que se implementam tais arquiteturas (Luthria e Rabhi, 2012). Segundo os autores deste artigo, uma arquitetura orientada a serviços não esconde a má engenharia de sistemas anteriores.

Também as infraestruturas baseadas em ambientes *Cloud* podem potenciar este tipo de arquitetura, sendo um ambiente natural para a implementação de uma arquitetura orientada a serviços.

Neste âmbito, a área das aplicações de gestão empresarial tem especial interesse na investigação de novas soluções para os seus produtos. Muitas destas empresas têm muitos anos de desenvolvimento de produtos sobre a mesma plataforma e têm maior interesse em soluções que visem facilitar a interoperabilidade da mesma com soluções mais recentes. Estas novas formas de desenvolvimento de aplicações, sob um paradigma arquitetural diferente, permitem gerar ideias inovadoras e abrem portas a novos mercados ainda em crescimento, podendo facilitar o rápido desenvolvimento de novas soluções.

2.1 Arquitetura orientada a Serviços

Ao longo dos anos têm surgido desenvolvimentos tecnológicos, como a arquitetura orientada a serviços, com o objetivo de ultrapassar as dificuldades apontadas.

Um dos principais objetivos de uma arquitetura orientada a serviços (Service Oriented Architecture – SOA) é alinhar o mundo dos negócios com o mundo das TIC, de uma forma que os torne a ambos mais eficientes (Rob High et al., 2005). A SOA é uma ponte que cria um relacionamento mais forte entre estes dois mundos de uma forma que ainda não tinha sido conseguida até então.

A OASIS é uma organização responsável por promover o desenvolvimento e a adoção de padrões em diversas áreas tecnológicas. A definição dada em OASIS (2006) para a SOA é a seguinte:

Arquitetura Orientada a Serviços (Service Oriented Architecture – SOA) é um paradigma para organizar e utilizar competências distribuídas que podem estar sob controlo de diferentes domínios proprietários.

No dia-a-dia, de uma forma simplista, tanto as pessoas como as organizações têm necessidades e problemas que pretendem resolver, e que podem ser resolvidos usando as competências oferecidas por outras pessoas ou organizações. Nem sempre existe uma relação de um para um entre necessidades e competências uma vez que a granularidade de ambas pode variar entre algo simples e algo muito complexo.

Um **serviço** é um mecanismo que permite aceder a uma ou mais capacidades usando uma interface prescrita (OASIS, 2006). Este serviço é providenciado por uma entidade, o fornecedor do serviço *(service provider)*, para ser usado por outros. No entanto, os eventuais consumidores do serviço podem não ser conhecidos pelo fornecedor do serviço. Uma das características dos serviços é a sua opacidade, no sentido de que a sua implementação está tipicamente escondida do consumidor do serviço *(service consumer),* exceto a informação que é requerida para saber se um serviço é apropriado para as suas necessidades.

A OASIS definiu ainda três conceitos fundamentais para perceber o que está envolvido na interação com serviços. Visibilidade entre fornecedores e consumidores de serviços, interação entre eles e o efeito da interação no mundo real (OASIS, 2006):

- **Visibilidade**: refere-se à capacidade daqueles com necessidades e aqueles com competências serem capazes de se verem uns aos outros. Isto é feito ao fornecer descrições e requisitos técnicos, restrições e outras políticas, e ainda mecanismos para acesso e resposta. Estas descrições devem estar num formato em que a sua sintaxe e semântica sejam amplamente acessíveis e compreensíveis;
- **Interação**: consiste na atividade de usar uma competência, normalmente através de uma série de trocas de informação e ações invocadas. Esta interação forma um caminho entre aqueles com necessidades e aqueles com competências.
- **Efeito**: o objetivo de usar uma competência é sempre obter um efeito. No fundo, uma interação é um ato e o resultado de uma interação é um efeito ou conjunto de efeitos. Este efeito pode ser o retorno da informação ou a alteração do estado de entidades conhecidas ou não-conhecidas envolvidas na interação.

O conceito principal que agrega todos os outros é o conceito de serviço, o qual é a base da SOA. Na SOA, os serviços são o mecanismo através do qual as necessidades e as

competências são ligadas, permitindo organizar as soluções de forma a promover a reutilização, escalabilidade e interoperabilidade (OASIS, 2006). Estes permitem providenciar abstrações de alto nível para organizar aplicações para ambientes abertos e de larga escala (Huhns e Singh, 2005).

Os conceitos de visibilidade, interação e efeito são também aplicados aos serviços da mesma forma que são aplicados ao paradigma SOA no geral. A visibilidade é promovida através de uma descrição do serviço, que contém a informação necessária para interagir com o serviço, e descreve o que será conseguido quando este é invocado. Para além disso, contém também as condições para usá-lo. Esta descrição permite que os potenciais consumidores do serviço decidam se este tem a competência necessária para cobrir as suas necessidades.

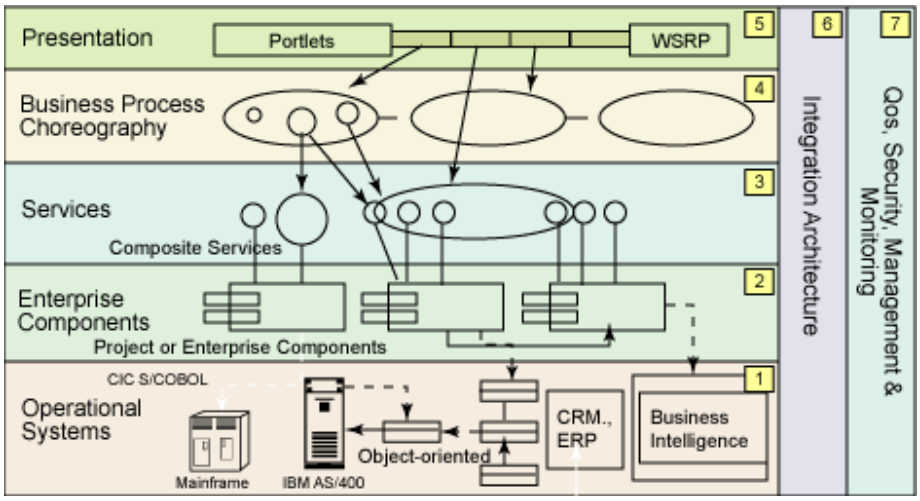
A SOA é uma forma de organização das soluções e não traz nenhum elemento do domínio que não exista sem SOA (OASIS, 2006). Outro aspeto importante é que existe flexibilidade entre os diversos intervenientes, isto é, podem existir entidades que tenham o conhecimento necessário para construir e evoluir uma certa competência, mas não estejam interessadas ou não tenham o conhecimento para construir o serviço de acesso a essas competências, permitindo a outras entidades assumirem esse papel. De referir também, que a arquitetura SOA é um paradigma totalmente agnóstico no que toca a tecnologia.

A característica que distingue uma abordagem orientada a serviços é a forma como possibilita a separação de responsabilidades, numa perspetiva que transcende os aspetos tecnológicos (Erl, 2005). Isto permite que grandes problemas possam ser resolvidos, usando uma melhor construção lógica, decomposta numa coleção de pequenas partes relacionadas, em que cada uma dessas partes "responde" a uma parte específica do problema. Este isolamento de responsabilidades não é novo. A programação orientada a objetos e, acima disso, a organização em componentes, preconiza exatamente isso, no entanto, continua a um nível baixo para que os intervenientes no negócio consigam perceber claramente as funcionalidades que se pretendem.

Na Figura 2.2 pode ver-se o exemplo de uma arquitetura SOA dividida em diversas camadas. Na camada número um encontra-se a camada dos sistemas operacionais. Esta camada é composta por aplicações feitas à medida e outro tipo de aplicações como aplicações de ERP ou CRM. Esta é a camada onde se encontra o *software* legado, o qual pode ser alavancado e integrado com sistemas mais atuais. Na segunda camada encontram-se componentes tipicamente implementados em servidores de aplicações *(application servers)* que

têm preocupações com características de qualidade de serviço como balanceamento de carga ou a alta disponibilidade.

Figura 2.2 – Camadas de uma Arquitetura SOA



Fonte: Arsanjani (2004)

A terceira camada é onde estão expostos os serviços para invocação e utilização direta ou uso em junção com outros serviços. Esta composição de serviços pode ocorrer numa quarta camada, podendo então ser usados como um só após essa composição. Na quinta camada encontra-se a camada de apresentação ao utilizador. A camada representada pelo número seis é a camada de integração, a qual existe em algumas tecnologias que tratam da integração de fim a fim. Relativamente à sétima camada, esta providencia as capacidades necessárias para monitorizar a qualidade do serviço relativamente à segurança, desempenho e disponibilidade.

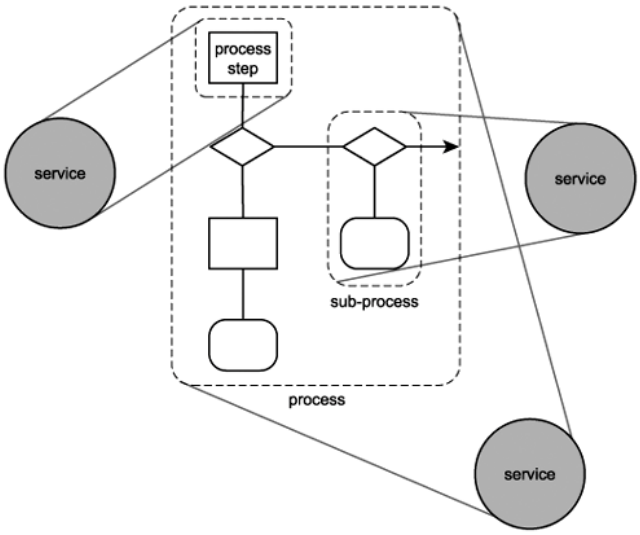
A vantagem da separação de responsabilidades, numa perspetiva orientada a serviços, é o grau de liberdade que permite ter na organização das mesmas, dando a possibilidade de agrupar aspetos comuns de negócio para benefício dos consumidores de serviços.

De forma a manter a sua independência, os serviços encapsulam a sua lógica dentro de um determinado contexto (Erl, 2005). Este contexto pode ser específico a uma tarefa, uma entidade de negócio ou outra organização lógica.

O exemplo dado por Erl (2005), consiste na automatização de um processo de negócio. A lógica de negócio é decomposta numa série de passos que executam numa sequência de acordo com as regras de negócio.

Na **Figura 2.3** pode observar-se um fluxograma que representa um processo de negócio onde se visualizam os papéis que um serviço pode assumir.

Figura 2.3 – Serviços



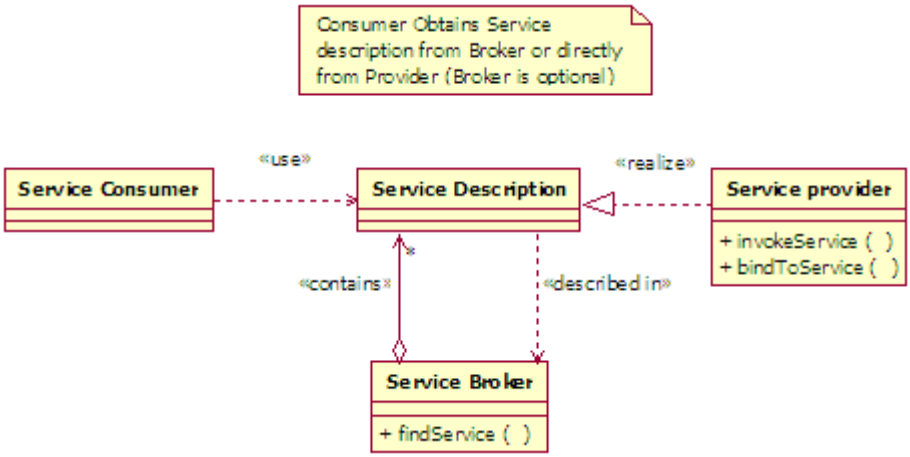
Fonte: Erl (2005)

Cada uma das tarefas pode ser encapsulada por um serviço, tanto individualmente como em conjunto, ou até no limite do processo inteiro. No fundo, cada serviço pode abranger diferentes quantidades de lógica, sendo que alguns serviços podem encapsular outros serviços de contextos mais específicos.

2.1.1 Modelo de interação

O autor Arsanjani (2004) apresenta também um modelo de interação entre os diferentes intervenientes (**Figura 2.4**). O fornecedor do serviço *(Service Provider)* publica a descrição do serviço e fornece a implementação do mesmo. O consumidor do serviço *(Service Consumer)* pode aceder diretamente à descrição do serviço, fornecida pelo fornecedor de serviço, isto caso tenha conhecimento da localização do mesmo através de um *Uniform Resource Identifier (URI).* Quando não tem conhecimento dessa localização terá que aceder a um "*Service Broker*" que o informará da localização da descrição. O "*Service Broker*" será um local/diretório de serviços onde os consumidores de serviços poderão aceder e encontrar serviços que pretendam consumir. Não é, no entanto, um interveniente obrigatório nesta interação.

Figura 2.4 – Modelo de Interação



Fonte: Arsanjani (2004)

Também Erl (2005) descreve a forma como os serviços se relacionam: *"... the relationship between services is based on an understanding that for services to interact, they must be aware of each other. This awareness is achieved through the use of service descriptions".* A descrição de serviço *(service description* ou *service contract)* é um conceito importante. Este expressa a interface de um serviço e pode ser composto por um ou mais documentos que expressam meta informação acerca de um serviço. No fundo servem como API *(Application Programming Interface)* para a funcionalidade oferecida por um serviço. A descrição de serviço tem como formato base uma estrutura composta pelo nome do serviço e pelos tipos de dados esperados e devolvidos pelo serviço. Esta forma de relacionamento entre serviços é classificada como de baixa dependência, uma das características fundamentais de uma arquitetura SOA. Outro aspeto importante é que os serviços comunicam através de mensagens e que as estas são unidades independentes de comunicação (Erl, 2005).

O autor Erl (2005) apresentou oito princípios base considerados essenciais para uma verdadeira arquitetura SOA:

- Baixa dependência: Os serviços devem manter um relacionamento que minimiza dependências e que requer apenas que eles retenham conhecimento uns dos outros;
- Contrato: Os serviços aderem a um acordo de comunicação, definido por um ou mais descrições de serviço;
- Autonomia: Serviços têm controlo sobre a lógica que encapsulam;
- Abstração: Para além do definido no contrato, serviços devem esconder a lógica do resto do mundo;

- Reutilização: A lógica é dividida em serviços com o objetivo de promover reutilização;
- Composição: Os serviços podem ser coordenados e combinados de forma a compor novos serviços compostos;
- Sem estado: Os serviços devem minimizar a retenção de informação específica a uma atividade;
- Capacidade de descoberta: Os serviços são desenhados para serem descritivos e para que possam ser encontrados e acedidos através de mecanismos de descoberta de serviços.

Nesta secção, foi feita uma primeira abordagem ao tema da arquitetura orientada a serviços numa perspetiva mais teórica. Foi feita uma descrição deste tipo de arquitetura e os objetivos que se pretendem atingir ao implementá-la. Este é um tipo de arquitetura que se pretende que permita organizar as soluções de forma a promover a reutilização, escalabilidade e interoperabilidade.

Foi também analisado o conceito principal deste tipo de arquitetura: o serviço. Foram analisados os conceitos do domínio e explicados os diversos papéis das peças intervenientes. Na próxima secção será seguida uma perspetiva mais tecnológica do tema dos serviços e de que forma podem ser implementados no âmbito desta dissertação.

2.2 Serviços Web

Na secção anterior, foi feito um estudo relativo aos conceitos fundamentais associados a uma arquitetura orientada a serviços. Nesta secção será abordada uma perspetiva mais tecnológica. Este tipo de arquitetura pode ser implementado usando diferentes metodologias, as quais serão analisadas de seguida.

A aplicação deste tipo de arquitetura, no âmbito da temática desta dissertação, é feita recorrendo a serviços web *(web-services)*, pelo que, o ambiente natural destes, como o próprio nome indica, é a *web*. Desde há muitos anos que o mundo tecnológico tenta ligar lógica de negócio através da rede. Várias tecnologias de sistemas distribuídos tentaram alcançar este objetivo, mas nenhuma conseguiu atingir o sucesso esperado. CORBA *(Common Object Request Broker Architecture)*, DCOM *(Distributed Component Object Model)*, RPC *(Unix Remote Procedure Call)*, e Java RMI *(Remote Method Invocation),* são talvez as mais conhecidas. Foram então desenvolvidos esforços no sentido de encontrar um padrão para estes serviços, atualmente responsabilidade da *World Wide Web Consortium (W3C),* um consórcio composto por empresas, órgãos governamentais e outras organizações, com a finalidade de estabelecer padrões para a criação e interpretação de conteúdos para a Web.

A W3C define serviço web da seguinte forma:

"A Web service is a software system designed to support interoperable machine-to-machine interaction over a network. It has an interface described in a machine-processable format (specifically WSDL). Other systems interact with the Web service in a manner prescribed by its description using SOAP messages, typically conveyed using HTTP with an XML serialization in conjunction with other Web-related standards."

A partir desta definição é possível encontrar algumas orientações na construção de serviços no entanto, a expressão serviço *web* tem sido usada com uma conotação menos restritiva relativamente à sua implementação. Como será analisado posteriormente, o desenho de serviços usando uma arquitetura REST *(Representational State Transfer)* tipicamente não recorre a mensagens SOAP nem a documentos WSDL, valendo-se, na maior parte dos casos, apenas do protocolo http *(Hypertext Transfer Protocol)*.

Na primeira parte desta subsecção serão introduzidos alguns conceitos chave no estudo dos serviços *web*. Na segunda parte será feita uma análise a diferentes arquiteturas de implementação de serviços *web*, sendo apresentadas as características de cada um e as vantagens e desvantagens de cada uma das abordagens.

2.2.1 Protocolos

Antes de serem demonstradas as diferenças entre os diversos tipos de arquiteturas, é importante fazer uma breve explicação de alguns conceitos chave na implementação de cada um dos tipos. Será feita uma breve análise ao protocolo http, ao protocolo SOAP e aos documentos WSDL.

http

O http é um protocolo da camada de aplicação no modelo OSI *(Open Systems Interconnection)*, um modelo que caracteriza e padroniza as funções de um sistema de comunicações ao particioná-lo em camadas de abstração diferentes.

No http, a aplicação cliente submete um documento num "envelope" a um servidor. O servidor envia para o cliente uma resposta também dentro de um "envelope". O protocolo tem regras rígidas relativamente ao formato que estes "envelopes" devem ter, apesar de não ter imposições acerca do seu conteúdo.

Listagem 2.1 – Um pedido http GET enviado ao *host* http://primaverabss.com

```
GET /corporate/Home-Home.aspx HTTP/1.1
Host: www.primaverabss.com
User-Agent: Mozilla/5.0
Accept: text/html
Accept-Language: pt-pt
Accept-Encoding: gzip, deflate
Connection: keep-alive
```

Na **Listagem 2.1** pode ver-se o exemplo de um envelope http. As principais propriedades de um pedido http são as seguintes:

- **Método**: Neste exemplo foi usado o método *GET*. Pedidos feitos usando *GET* devem ser usados apenas para pedir dados ao servidor. O protocolo define, além do *GET*, um conjunto de métodos (*POST*, *DELETE*, *PUT*, etc.) para indicar a ação desejada a ser executada no recurso identificado. Cada um dos métodos tem uma semântica associada e o servidor pode despoletar ações diferentes consoante o método definido.

- **Path**: É a parte do URI à direita do *host:* neste exemplo, o *path* de "www.primaverabss.com/corporate/Home-Home.aspx" é "/corporate/Home-Home.aspx". O *path* é o endereço no *host* para onde o envelope *http* é enviado.
- **Cabeçalho do Pedido**: no cabeçalho do pedido encontram-se pequenos pedaços de metadados que podem ajudar na comunicação (e.g. *Accept, Accept-Language*)
- **Corpo do envelope**: no corpo do envelope pode ir o documento que se pretende enviar. Neste caso, sendo um pedido *GET*, o corpo vai vazio que é o mais comum neste tipo de pedidos.

Na resposta a este pedido é também enviado um documento num envelope *http* (**Listagem 2.2**).

Listagem 2.2 – Resposta http a um pedido GET

```
HTTP/1.1 200 OK
Connection: Keep-Alive
Proxy-Connection: Keep-Alive
Content-Length: 43902
Date: Tue, 02 Jul 2013 15:01:03 GMT
Content-Type: text/html; charset=utf-8
Server: Microsoft-IIS/7.0
Cache-Control: private
X-AspNet-Version: 2.0.50727
X-Powered-By: ASP.NET


<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head id="Head1">
<title>
       PRIMAVERA BSS Corporate - Home - Powered by PRIMAVERA WebCentral
(…)
```

Esta resposta pode ser dividida em três partes:

- **Código *http* da resposta:** é um código numérico que indica às aplicações cliente se o pedido foi bem ou mal sucedido. O protocolo define um conjunto de códigos com semânticas diferentes de forma a possibilitar que os clientes os interpretem de forma diferente. Na **Listagem 2.2** o código enviado foi o **"200 OK"** que indica que o pedido foi bem-sucedido.
- **Cabeçalhos da resposta:** tal como no cabeçalho do pedido, inclui meta-dados informativos *(Content-Type, Content-Length, Server, etc.)*

- **Corpo do envelope:** Mais uma vez, tal como no corpo do pedido, é no corpo do envelope que segue o recurso requisitado no pedido *GET*. Na **Listagem 2.2** o corpo do envelope é composto pelo código HTML *(HyperText Markup Language)*.

Nos cabeçalhos da resposta existe uma propriedade com maior importância que as restantes: *Content-Type*, neste caso *text/html*. É esta propriedade que permite ao navegador *web* saber que pode reproduzir o conteúdo do corpo do envelope como uma página *web*. Tal como nos códigos *http* da resposta, também existe uma lista de *content-types* para conteúdos comuns, como documentos de dados estruturados *(application/xml)* ou imagens *(image/jpeg)*.

SOAP

O protocolo *SOAP* é um protocolo para a troca de informação estruturada, usado na implementação de serviços web. A sua estrutura é baseada no formato XML e utiliza outros protocolos da camada de aplicação como o *http* ou o *smtp* para a transmissão das mensagens. Este foi desenhado para ser independente de qualquer modelo de programação (Gudgin, 2007).

Apesar de não existirem regras neste sentido, é comum uma mensagem SOAP levar as informações necessárias para executar um método específico do servidor. A interação entre cliente e servidor é baseada num contrato previamente definido e exposto num documento *WSDL (Web Services Description Language)*.

Uma mensagem SOAP (**Listagem 2.3**), como foi dito, tem uma estrutura em XML com os seguintes elementos:

- ***Envelope:*** Identifica o documento como sendo uma mensagem SOAP. Os outros dois elementos são filhos deste.
- ***Header:*** Contém a informação do cabeçalho e é um elemento opcional. Nele podem encontrar-se informações adicionais que possam ser necessárias na comunicação da mensagem. Não existem elementos obrigatórios para este conteúdo.
- ***Body:*** Este é o corpo da mensagem e contém os dados específicos desta. Os elementos do corpo são específicos de cada implementação e incluem tipicamente métodos a invocar e respetivos argumentos.

Listagem 2.3 – Mensagem SOAP

```
<env:Envelope xmlns:env="http://www.w3.org/2003/05/soap-envelope">
 <env:Header>
  <n:alertcontrol xmlns:n="http://example.org/alertcontrol">
   <n:priority>1</n:priority>
   <n:expires>2001-06-22T14:00:00-05:00</n:expires>
  </n:alertcontrol>
 </env:Header>
 <env:Body>
  <m:alert xmlns:m="http://example.org/alert">
   <m:msg>Pick up Mary at school at 2pm</m:msg>
  </m:alert>
 </env:Body>
</env:Envelope>
```

Fonte: Gudgin (2007)

O protocolo foi desenhado com o objetivo principal de providenciar uma *framework* de troca de mensagens, mas também com objetivo de providenciar mecanismos de extensibilidade, que permitem adicionar funcionalidades mais ricas como fiabilidade e segurança.

Existe um conjunto de especificações/padrões conhecidos como "WS-*" aplicados aos serviços SOAP. Algumas das especificações são as seguintes:

- **WS-Security:** especificação para aplicar mecanismos de segurança à troca de mensagens SOAP. A especificação determina de que forma pode ser garantida a integridade e a segurança das mensagens e providencia segurança de fim a fim.
- **WS-Addressing:** providencia os mecanismos necessários para que o endereçamento de mensagens e serviços seja completamente independente do protocolo de transporte subjacente.
- **WS-AtomicTransaction:** especificação que garante que a propriedade de transação (tudo ou nada) seja implementada.
- **WS-ReliableMessaging:** protocolo que define um conjunto de mecanismos para que a troca de mensagens seja fiável.

Em resumo, o protocolo SOAP define um conjunto de convenções para trocar mensagens XML. Este protocolo permite a troca de dados estruturados entre pares, num ambiente descentralizado e distribuído.

WSDL

WSDL é o acrónimo para *Web Services Description Language*, ou seja, é um documento que descreve um serviço *web.* Este é escrito em XML e especifica a localização do serviço e os métodos que o serviço expõe, funcionando no fundo, como contrato do serviço.

Um documento WSDL (**Listagem 2.4**) tipicamente é composto pelos seguintes elementos:

- ***Types:*** é dentro deste elemento que são definidos os tipos de dados usados neste serviço.
- ***Message:*** contém a definição dos dados a serem trocados. Cada elemento pode ter um ou mais subelementos chamados *parts.* Estas podem ser comparadas aos parâmetros de uma função.
- ***PortType:*** conjunto de operações suportadas por um ou mais *endpoints.* Cada *portType* pode ser comparada a uma classe numa linguagem de programação tradicional. Nela são definidas as operações que podem ser executadas e as mensagens envolvidas nessas operações.
- ***Binding:*** especificação particular do protocolo e formato de dados a usar numa porta em particular.

Listagem 2.4 – Fração de um documento WSDL

```
<message name="getTermRequest">
  <part name="term" type="xs:string"/>
</message>

<message name="getTermResponse">
  <part name="value" type="xs:string"/>
</message>

<portType name="glossaryTerms">
  <operation name="getTerm">
    <input message="getTermRequest"/>
    <output message="getTermResponse"/>
  </operation>
</portType>
```

Fonte: (w3schools)

Os documentos WSDL definem o contrato de um serviço *web.* Permitem gerar automaticamente o código necessário no lado dos consumidores para implementar uma dada interface. Este tipo de documentos estão normalmente associados a serviços *web* que usam mensagens SOAP como forma de comunicação.

Nem todos os tipos de serviços necessitam de mensagens SOAP e documentos WSDL. Como será explicado na secção seguinte existem formas de construir serviços *web* sem recorrer a estes mecanismos.

2.2.2 Arquiteturas de Serviços Web

A implementação de serviços *web* foi dividida por Richardson e Ruby (2007) em três tipos de arquiteturas: *RESTful*, *RPC (Remote Procedure Call)* e um híbrido *REST-RPC*. Nesta secção serão apresentadas as características de cada um dos tipos.

RESTful

O *REST (Representational State Transfer)* é um conjunto de princípios arquiteturais para a construção de sistemas distribuídos. O termo foi apresentado na dissertação de doutoramento de Roy Fielding (Fielding, 2000), um dos autores do protocolo *http*, que é fundamental nesta arquitetura. Quando uma arquitetura aplica os princípios do REST diz-se que é *RESTful*.

A característica fundamental do REST é o uso do *http* como sistema de transporte para interações remotas. Fielding na sua dissertação descreveu como os sistemas de informação distribuídos, como a *Web*, são construídos. Este descreveu a forma de interação entre os diversos recursos e como se relacionam através de identificadores únicos (URI).

Numa arquitetura REST existem dois conceitos chave que devem ser explicados: **métodos** e **nomes**. O objetivo deste tipo de arquitetura é aproveitar todas as potencialidades do protocolo http. Como já foi referido, o protocolo http define um conjunto de métodos. Alguns dos mais importantes são de seguida definidos de uma forma simplista:

- **GET:** usado para obter dados;
- **PUT:** sobrescrever com dados diferentes;
- **DELETE:** apagar dados;
- **POST:** usado na criação de novas entidades e em outras operações.

Esta característica pode ser uma vantagem, uma vez que a utilização destes métodos/verbos segue uma forma padrão, com semânticas claras e suficientemente abrangentes para as operações mais comuns.

A outra parte importante do REST está relacionada com os nomes. Os nomes indicam os recursos sob os quais podem ser efetuadas as ações definidas pelos métodos *http*. É desta forma que as aplicações cliente comunicam ao servidor sob que dados querem que a ação ocorra.

Listagem 2.5 – Pedido fictício ao recurso /clientes

```
GET /clientes HTTP/1.1
Host: www.primaverabss.com
Accept: application/json
```

Na **Listagem 2.5** pode ver-se um pedido fictício. O pedido define o nome sobre o qual se pretende efetuar a ação, neste caso no recurso "/clientes". O método é o *GET*, pelo que um pedido enviado a este endereço irá devolver uma representação da lista de clientes. Um dos cabeçalhos da mensagem *http* é a propriedade *Accept* que indica *"application/json"*. Isto indica o tipo de dados que a aplicação cliente espera obter, permitindo ao servidor devolver a resposta adequada. Na resposta (**Listagem 2.6**) a este pedido o servidor envia uma mensagem *http* semelhante à definida na **Listagem 2.2**, mas em que o corpo da mensagem inclui o estado atual da representação existente em "/clientes".

Listagem 2.6 – Resposta ao pedido

```
HTTP/1.1 200 OK
Cache-Control: no-cache, no-store
Content-Length: 561
Content-Type: application/json; charset=utf-8
Server: Microsoft-IIS/8.0
X-AspNet-Version: 2.0.50727
X-Powered-By: ASP.NET
Date: Fri, 25 Oct 2013 13:46:33 GMT
[
   {
      "ClientInfo":{
         "Id":"829f54e8-b8a8-411a-9cbc-73a081e6cdba",
         "Login":"andre_couto_silva@example.com",
         "Name":"André Silva"
      },
   },
   {
      "ClientInfo":{
         "Id":"6ae11ea7-4e3a-498f-9415-6378688166dc",
         "Login":"jorge_carvalho@example.com",
         "Name":"Jorge Carvalho"
      },
   }
]
```

Segundo Roy Fielding, uma arquitetura verdadeiramente RESTful segue o conceito de HATEOAS – Hypermedia as the Engine of Application State. De acordo com este princípio, as aplicações cliente não necessitam de conhecer previamente todos os URI's dos recursos do sistema, mas apenas um ponto de partida. Todas as futuras ações que o cliente possa tomar são descobertas a partir das representações dos recursos devolvidas pelo servidor. Estas são acompanhadas de uma descrição do relacionamento que têm com a representação anterior. Assim, a interação é dirigida pelo hipertexto e não por informações externas, como a documentação de uma API. O benefício mais claro à primeira vista é a possibilidade do servidor alterar os URIs dos recursos sem quebrar os clientes (Fowler, 2010), mas Fielding defende que este conceito é parte integrante do que se considera uma arquitetura que segue os princípios REST, sendo essencial na escalabilidade e longevidade do *software*. (Fielding, 2008)

Nem sempre os princípios definidos por Fielding são totalmente aplicados na implementação da arquitetura do serviço, o que normalmente implica que o conceito de HATEOAS não é aplicado. Este é o conceito mais complexo de implementar, tanto no serviço como nas aplicações clientes, sendo no entanto aquele que garante maior longevidade do serviço. Quando este conceito não é aplicado, é normalmente necessário ter um documento com uma API pública que os programadores terão que ler para implementar as aplicações cliente.

Remote Procedure Call

Numa arquitetura do tipo RPC *(Remote Procedure Call)*, cada serviço tem um novo vocabulário. Da mesma forma que um programador define novas funções/métodos com novos nomes quando constrói um novo programa, numa arquitetura RPC, cada serviço tem também novos métodos. A diferença óbvia face a um serviço RESTful é que nestes, todo o tipo de entidades responde ao mesmo conjunto de métodos (GET, PUT, DELETE...).

O percursor do protocolo SOAP – XML-RPC – é o exemplo mais óbvio de uma arquitetura deste tipo.

Listagem 2.7 – Documento XML de um pedido RPC

```
<?xml version="1.0" ?>
<methodCall>
  <methodName>lookupUPC</methodName>
  <params>
    <param><value><string>001441000055</string></value></param>
  </params>
</methodCall>
```

Fonte: Richardson e Ruby (2007)

Neste protocolo também o documento XML é enviado através de *http*, no entanto, ignora a maior parte das funcionalidades do protocolo *http*. Todos os pedidos são enviados tipicamente para um único endereço e usando apenas o método POST. O servidor apenas lê os documentos e executa o método invocado.

Arquitetura Híbrida

Os autores Richardson e Ruby (2007) apresentam ainda um tipo de arquitetura híbrida. Segundo estes, este tipo de arquitetura acontece quando os princípios do REST não são totalmente aplicados. Tipicamente, numa arquitetura híbrida, o protocolo *http* é usado apenas como envelope, enquanto o URI é usado como método a invocar (e.g. http://example.com/getPhotos).

Comparação dos tipos de arquiteturas

Uma arquitetura RPC está em norma associada ao uso do protocolo SOAP como forma de comunicação, enquanto uma arquitetura RESTful regra geral usa o protocolo *http.*

Em arquiteturas RPC é necessário recorrer ao protocolo WSDL para definição da interface de utilização. A criação e manutenção dos documentos WSDL é tipicamente uma tarefa complexa pelo que é normalmente da responsabilidade de ferramentas de geração automática, que interpretam os contratos definidos pelos programadores e geram o documento WSDL. Na Listagem 2.8 pode ver-se como se pode expor um método usando a linguagem de programação C#.NET.

Para fazê-lo é necessário anotar o método com uma etiqueta que indique que é uma operação disponível *(OperationContract)* no contrato do serviço.

Listagem 2.8 – Exemplo de uma interface C# que expõe uma operação

```
[ServiceContract]
public interface IService1
{
   [OperationContract]
   string GetData(int value);
}
```

Este processo abstrai também uma série de configurações necessárias para que o protocolo funcione. O documento WSDL reflete o contrato relativo a estas operações. As aplicações cliente acedem ao documento WSDL e têm que interpretá-lo de forma a perceber como podem comunicar com o serviço. A dificuldade deste processo está muito dependente das ferramentas usadas no mesmo, o que pode trazer alguns problemas. O primeiro é precisamente a dependência das ferramentas no processo. Uma das premissas de uma arquitetura orientada a serviços é o aumento da interoperabilidade contudo, a dependência de ferramentas para consumir um serviço é um aspeto que não é desejável. Outro problema deste método é a dependência que existe entre as aplicações cliente e o servidor. Ao associar diretamente métodos da lógica de negócio aos serviços, as aplicações cliente ficam muito dependentes do contrato WSDL, quebrando à mínima alteração do mesmo. Qualquer alteração nesta lógica pressupõe que todas as aplicações cliente têm que ser atualizadas.

Escalabilidade, Fiabilidade e Segurança em REST

A implementação de qualquer arquitetura de um sistema de *software* deve ter sempre em atenção um conjunto de propriedades desejáveis, sendo que a importância de cada uma das propriedades vai depender do problema em questão. Nesta subsecção serão discutidos aspetos de escalabilidade, fiabilidade e segurança em arquiteturas REST.

Fiabilidade: um dos pontos em que uma arquitetura REST pode levantar algumas dúvidas é a fiabilidade. Numa arquitetura SOAP é possível implementar o protocolo *WS-ReliableMessaging,* que inclui mecanismos para garantir que a troca de mensagens não falha, nomeadamente usando os seguintes padrões de interação: *in order, at least once, at most once, exactly once*.

Usando *http*, é possível atingir os mesmos requisitos através dos verbos *http (GET, PUT, etc.)*, dos cabeçalhos das mensagens e dos códigos de retorno *http* (Webber et al., 2010). Uma das formas de aumentar os índices de fiabilidade é através de operações idempotentes. Uma operação idempotente é aquela que pode ser executada várias vezes e que irá obter o mesmo resultado todas as vezes. As operações que usam os métodos GET, PUT, DELETE são idempotentes. Quando o pedido original falha, basta repeti-lo para obter o mesmo resultado, sem efeitos secundários. Os códigos de retorno permitem que as aplicações cliente sigam diferentes ações consoante o código retornado. A exceção a esta regra acontece com o método POST. O método POST é usado para criar novos recursos, pelo que é necessário que a lógica de negócio implemente o mecanismo necessário para garantir que a repetição de uma ação não traz resultados inesperados.

Escalabilidade: um dos pontos mais fortes de uma arquitetura REST é a relativa facilidade com que é possível escalar uma arquitetura de forma a ser utilizada por milhões de dispositivos clientes. Numa arquitetura REST o servidor não guarda o estado da aplicação, permitindo facilmente aumentar a escalabilidade horizontal – aumentando o poder de processamento do servidor. Outra característica que uma arquitetura REST permite aproveitar, é a implementação de mecanismos de *caching*. O *caching* consiste na habilidade de guardar cópias de dados que são acedidos frequentemente em diversos locais. Estes mecanismos podem aliviar a carga a que o servidor principal está sujeito, assim como o tráfego de dados usado na rede, podendo até reduzir a latência dos pedidos. A introdução do *caching* pode, no entanto, trazer problemas de consistência entre os dados do servidor e os dados dos consumidores, pelo que deve ser usado com critério e com recurso a outras técnicas, para garantir que os dados se mantêm "frescos". O estado de uma aplicação deve ser mantido na aplicação e os pedidos feitos ao servidor devem ter toda a informação necessária para garantir a resposta ao pedido.

Segurança: numa arquitetura SOAP é possível implementar protocolos como o WS-Security e os protocolos associados WS-Federation, WS-SecureConversation ou WS-Trust, como forma de garantir a segurança. Estes utilizam um conjunto de técnicas criptográficas de maneira a providenciar segurança de fim a fim na transferência de mensagens. Apesar de aparentemente a utilização destes protocolos ser inquestionável, em (Webber et al., 2010) é defendido que os sistemas mais simples têm tendência a ser mais seguros, dado que existem menos locais onde possam existir falhas, oferecendo menos oportunidades para ataques maliciosos. Os autores

defendem ainda que a sofisticação da pilha WS-Security traz um custo em termos de complexidade, que leva a que mais uma vez exista a dependência de ferramentas, de forma a esconder essa complexidade. Ao esconder certos aspetos do sistema aos programadores é mais fácil que os mesmos cometam erros, acabando por tornar o sistema menos fiável.

A *web* não possui um mecanismo de segurança tão sofisticado, mas é possível encontrar outros mecanismos que aumentam a segurança de um sistema. O protocolo http suporta a autenticação cliente-servidor com nome de utilizador e palavra-chave, tanto básica como usando uma aproximação *digest* (método que aplica uma função *hash* à palavra-chave). Existem ainda outros sistemas de autenticação, como o sistema OpenID, ou de autorização, como o OAuth.

A utilização de https*,* que recorre ao protocolo TLS/SSL, permite encriptar o transporte das mensagens, sendo este um dos mecanismos mais usados para garantir a segurança das comunicações. A utilização deste protocolo traz a desvantagem de impedir o mecanismo de *caching,* uma vez que os meta-dados http apenas estão disponíveis para o cliente e para o servidor, e não para a estrutura subjacente. A não utilização do mecanismo de *caching* reduz a escalabilidade de um sistema.

Como foi demonstrado, a construção de serviços web pode seguir diferentes paradigmas e protocolos. Nesta secção foram abordados basicamente dois paradigmas distintos: por um lado, um paradigma mais orientado à web e que aproveita todas as potencialidades do protocolo http e os conceitos que foram desenvolvidos ao longo dos anos na *World Wide Web.* Por outro, um paradigma orientado à invocação de métodos/funções, que resulta da herança da programação tradicional de construção de programas e que recorre a mensagens SOAP para garantir a interação, relegando o protocolo *http* a mecanismo de transporte. Cada um dos paradigmas tem as suas vantagens e desvantagens, pelo que a escolha da sua utilização terá que ter em conta uma série de fatores, que serão enunciados ao longo desta secção.

2.3 Modelos de implementação de ERP e a Computação na Nuvem

Nesta secção pretende-se apresentar, de uma forma geral, em que consiste um ERP e de que forma este tipo de aplicações informáticas tem evoluído.

Os ERP (Enterprise Resource Planning) são um tipo de aplicação informática usados maioritariamente por organizações. Segundo a Gartner[2], a definição de ERP é a seguinte:

"Enterprise resource planning (ERP) is defined as the ability to deliver an integrated suite of business applications. ERP tools share a common process and data model, covering broad and deep operational end-to-end processes, such as those found in finance, HR, distribution, manufacturing, service and the supply chain. (...) ". Um ERP pode então ser definido como uma aplicação que integra todos os dados e processos de uma organização num único sistema, facilitando a automatização de processos.

Atualmente existem várias formas de implementar sistemas ERP. Em (Beaubouef, 2012) é feita uma divisão das formas de implementação destes em quatro tipos distintos: sistemas locais, sistemas alojados, nuvem *(cloud)* pública e privada.

A implementação mais tradicional é feita em sistemas locais. Nestes, as organizações que utilizam o ERP detêm o *software* e o *hardware* necessário para sustentar o sistema. Este tipo de implementação implica normalmente mais investimento inicial, mas também permite maior controlo sobre o sistema. Outro tipo de implementação é o modelo de alojamento na *web*, em que o *software* é detido pelas organizações, mas o *hardware* e a infraestrutura são alugados por outra empresa. O modelo de implementação *cloud* é um modelo em que o tanto o *software* como o *hardware* são detidos pelos fabricantes de ERP, sendo que numa *cloud* pública, os recursos de *hardware* são partilhados com outros clientes, enquanto numa *cloud* privada, existe a possibilidade de ter recursos dedicados.

[2] Http://www.gartner.com/it-glossary/enterprise-resource-planning-erp/

2.3.1 Computação na Nuvem

Nesta secção será feita uma abordagem à computação na nuvem *(cloud computing),* à sua relação com uma arquitetura orientada a serviços e à forma como este modelo pode ser aplicado em aplicações empresariais.

A área da computação na nuvem é uma das áreas tecnológicas que tem estado sob foco nos últimos anos. A definição de *Cloud Computing,* dada pelo NIST *(National Institute of Standards and Technology)* em (Mell et al., 2011) é a seguinte:

"Cloud computing is a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction."

No fundo, o modelo de computação na nuvem consiste numa solução em que todos os recursos computacionais (hardware, software, *networking*, armazenamento, entre outros) são providenciados aos utilizadores numa lógica "*on-demand*", através da rápida realocação de recursos (Amrhein e Quint, 2009).

Para aproveitar as vantagens da computação na nuvem é necessária uma arquitetura que as suporte, tal como a arquitetura orientada a serviços. Na realidade a própria infraestrutura *cloud* é baseada nos princípios da SOA (Krill, 2009), neste caso aplicando esses princípios numa arquitetura de *deployment*.

A utilização de serviços que podem ser partilhados e reutilizados permite converter aplicações tipicamente verticais, em componentes como serviços, que podem ser reutilizados noutras aplicações, providenciando assim a agilidade necessária para fazer alterações mais rapidamente. Estas características fazem do modelo de computação na nuvem um ambiente natural para uma arquitetura orientada a serviços. A SOA providencia a organização necessária que permite que diferentes tipos de aplicações possam aceder facilmente a serviços cloud (Bowen, 2009).

Para melhor perceber a problemática das soluções empresariais na nuvem, foi estudada uma análise SWOT, a qual se poderá ver de seguida.

Análise SWOT

Os autores Marston et al. (2011) fizeram uma análise SWOT (Strengths, Weaknesses, Opportunities, Threats) acerca de soluções empresariais na *Cloud* que aqui será exposta.

Fraquezas

Um dos problemas apontados é a desconfiança por parte das empresas em permitir que todo o arquivo de dados das mesmas deixe de estar em seu poder localmente, para ficar alojado num local desconhecido ou em que tenham pouca confiança. Outro problema é que as grandes empresas têm também níveis de exigência muito elevados relativamente à qualidade e disponibilidade de serviço. Apesar de muitas soluções locais por vezes terem também os mesmos ou outros tipos de problemas, as soluções na nuvem são ainda vistas com desconfiança.

Forças

Não obstante aos problemas apontados, as soluções empresariais na nuvem têm também vantagens. Tirando partido da filosofia *cloud,* as pequenas empresas e empresas em início de atividade, não necessitam de investir em infraestruturas informáticas de elevado custo para instalarem as aplicações. Outra vantagem é a possibilidade de oferecer um novo tipo de aplicações e um conjunto de novos serviços que não eram possíveis anteriormente, abrindo, assim, portas a uma maior inovação. No artigo de Marston et al. (2011) encontramos o exemplo de aplicações móveis interativas que são conscientes da localização, ambiente e contexto que as rodeiam e, ainda, que respondem, em tempo real, à informação providenciada por utilizadores humanos, sensores não-humanos (sensores de humidade por exemplo) ou, por serviços de informação independentes (dados meteorológicos por exemplo).

Oportunidades

Os autores Marston et al. (2011) apontam como uma das principais oportunidades o potencial que estas soluções têm em países em desenvolvimento. A possibilidade de implementar soluções empresariais sem a necessidade de um avultado investimento inicial em infraestruturas, abre portas a novos mercados até então arredados do mapa. Outra oportunidade, já referida na secção anterior, prende-se com o exemplo das pequenas empresas em início de atividade.

Ameaças

Enquanto algumas empresas, principalmente de pequena dimensão, têm menos receio em dar um passo em direção a estas soluções, aproveitando as novas oportunidades que se abrem, outras empresas, principalmente as de grande dimensão, têm ainda muitas dúvidas de que as soluções na nuvem lhes consigam oferecer as mesmas condições que as soluções locais. Segurança, performance e fiabilidade, são pontos-chave para as grandes empresas em que estas soluções têm ainda dificuldades em competir com as soluções locais, apesar da rápida evolução tecnológica que têm conseguido nos últimos anos.

2.3.2 ERP híbridos

Com o advento das infraestruturas na nuvem, existe uma clara tendência no mercado dos ERP para uma adoção crescente de soluções mistas, denominadas 'híbridas'. Estas soluções conjugam soluções na nuvem com os sistemas locais, cuja migração para o paradigma *cloud* coloca ainda desafios técnicos e organizacionais e, por isso, não são equacionadas pelas empresas para curto prazo.

A implementação de ERP híbridos possibilita uma mistura entre os diversos modelos apresentados anteriormente, abrindo portas a novas oportunidades. Uma solução que suporta uma implementação híbrida tem de ser arquitetada de forma a suportar ambientes multiplataforma em simultâneo (Beaubouef, 2012).

Segundo Beaubouef (2012), o verdadeiro ERP híbrido é a solução de *software* que permite ao cliente a flexibilidade para implementar tanto os módulos como as funcionalidades principais através de múltiplas plataformas. Numa solução deste tipo, o processo de negócio pode atravessar os diversos modelos de implementação, aproveitando as vantagens de cada um, em aspetos diferentes do processo. Uma arquitetura orientada a serviços pode funcionar como facilitador na implementação de um modelo híbrido graças às características de interoperabilidade entre diferentes sistemas.

2.4 Aplicações da Loja Windows

O desenvolvimento de aplicações para diferentes dispositivos móveis como *smartphones* ou *tablets* é uma área em grande destaque nos últimos anos. A forma como se interage com as máquinas nunca teve tantas ofertas como atualmente. A multiplicidade de formatos de dispositivos, com vários tamanhos e formas de interação, nunca foi tão grande, avolumando-se as ofertas de diferentes multinacionais de *software.* A Gartner[3] prevê que em 2020 os utilizadores irão gastar menos de 10% do seu tempo em aplicações *desktop* tradicionais.

A proposta mais recente da Microsoft é o sistema operativo Windows 8/RT. A Infosys, uma das maiores multinacionais indianas da área das TIC, realizou um relatório (Sirangi e Bajpai, 2012) com as considerações chave a ter em conta na migração de aplicações existentes. No âmbito desta dissertação, importa retirar as considerações arquiteturais que as aplicações desenvolvidas para Windows 8/RT, chamadas de Aplicações da Loja Windows *(Windows Store Apps)*, apresentam como novidade, as quais permitirão integrar a aplicação com o trabalho em causa.

- **Camada de acesso a dados e de negócio**: Não existe suporte nativo para usar bases de dados neste tipo de aplicações. O acesso à camada de dados deve ser feito recorrendo a serviços web, por exemplo.
- **Sempre ligado:** Providenciar conteúdo atualizado através de notificações e *"live tiles"*. Ambas podem ser atualizadas a qualquer altura através do *"Windows Push Notification Service".*
- **Ligação à *Cloud:*** os utilizadores têm a possibilidade de sincronizar as suas aplicações, dados, fotografias, definições, etc., por todos os seus dispositivos.

No âmbito da presente dissertação, existem algumas características das Windows Store Apps que serão interessantes de investigar e relacionar com as temáticas abordadas nas secções anteriores e que se detalham de seguida.

[3] Http://www.gartner.com/newsroom/id/2061815

2.4.1 Experiência de Utilização

A Microsoft definiu um conjunto de guias de Experiência de Utilização *(User Experience)* (UX), que aconselham os programadores a seguir, e para os quais as suas aplicações estão orientadas. Existe um conjunto de novidades, face às aplicações tradicionais, a que se deve ter atenção na construção de uma *Windows Store App.* São aqui apresentadas algumas das características destas novas aplicações:

- **Interação através do toque:** as Windows Store Apps devem ser construídas tendo em mente que o utilizador pode interagir com as mesmas usando ecrãs sensíveis a toque. Existe um conjunto de características que podem ser implementadas para aproveitar esta possibilidade.
- **Layouts flexíveis:** a aplicação deve ser desenhada para funcionar em diferentes tipos de *layouts.* Não só os ecrãs podem rodar (como no caso dos *tablets),* mas também o próprio sistema operativo suporta um conjunto de diferentes tamanhos para cada aplicação.
- **Contratos e Atalhos:** os contratos são declarações que indicam que a aplicação tem suporte para uma certa funcionalidade. No Windows 8/RT existe uma barra de atalhos para alguns contratos comuns. O contrato de pesquisa é um exemplo comum. Se uma aplicação suportar o contrato de pesquisa, um utilizador pode utilizar a pesquisa da barra de atalhos do sistema para pesquisar na aplicação. Isto permite uma experiência de utilização consistente em todas as aplicações, possibilitando aos utilizadores adaptarem-se facilmente a uma interface.
- **Mosaicos e Notificações:** as *Windows Store Apps* usam mosaicos em vez dos tradicionais atalhos. Isto permite que estes mosaicos contenham informações atualizadas e notificações da aplicação, sem que o utilizador tenha que executar a aplicação propositadamente.

Estas são algumas das principais características das *Windows Store Apps* e que representam uma novidade face ao desenvolvimento típico de aplicações para computadores pessoais.

Este tipo de aplicação, além de seguir uma tendência no desenvolvimento de *software*, permite verificar a validade de uma arquitetura orientada a serviços, no cenário âmbito deste estudo.

Capítulo 3

Análise de Requisitos do Caso de Estudo

Neste capítulo será feita uma descrição do problema que é âmbito desta dissertação. Deste problema foram levantados os requisitos e feita uma análise, que se apresenta de seguida.

Para uma primeira discussão de requisitos foi criado um cenário e respetivo *storyboard,* com uma possível interação usando o sistema a desenvolver. Este cenário facilitou a identificação de funcionalidades necessárias.

3.1 Cenário

Nesta secção será apresentado o cenário de utilização desenhado.

"O Gonçalo tem uma pequena empresa de representação de marcas de luxo de produtos de cafetaria e bar, a Choc&Drink, Lda.. O Mário trabalha na Choc&Drink: é ele que gere o armazém, os *stocks* e faz as encomendas às empresas representadas. A Joana é secretária da administração numa grande empresa do sector dos transportes aéreos, a FlyBy S.A.

Perto da altura do Natal, o Gonçalo recebe uma série de encomendas para o fornecimento de produtos de oferta, que lhe é feita por diversas secretárias da administração e enviada por correio eletrónico. A maior parte destes pedidos são caixas de chocolates e utensílios de bar, que servem certamente para oferecer aos melhores clientes e fornecedores da FlyBy. Uma vez que em anos anteriores se instalou a confusão na FlyBy porque cada Administrador procedeu à sua encomenda em separado, com diferentes números de pedido, e dando azo a ocasionais duplicações de ofertas e posteriores devoluções, desta vez o Gonçalo pediu à FlyBy para coordenar a encomenda num único número de pedido. A Joana é que ficou de coordenar os pedidos com as outras secretárias e proceder à encomenda.

O Gonçalo, logo que recebe o pedido, trata de o introduzir no ERP e enviar a Nota de Encomenda ao Mário, através da nova aplicação - a NotesERP. Esta nota de encomenda é visualizada pelo Mário, que verifica uma referência no pedido que já não existe - uma caixa de 12 chocolates que fazia parte do catálogo do ano anterior e que saiu por engano no catálogo deste ano. O Mário junta uma nota dirigida ao Gonçalo, que é ligada ao campo de referência em causa: *"Gonçalo, como o erro foi nosso, sugiro o envio de mais uma caixa de 24 pelo preço da caixa de 12".* O Gonçalo concordou com a sugestão do Mário mas, com medo das confusões de anos anteriores, considerou que seria melhor a Joana verificar os pedidos antes de emitir uma fatura. O Gonçalo resolve, uma vez mais, utilizar a nova aplicação de envio de notas do ERP Primavera. Ele emite uma fatura em rascunho e partilha com a Joana, fazendo isso a partir do ERP. O Gonçalo inclui uma nota a pedir à Joana para conferir as quantidades e os artigos pedidos.

"Joana queira conferir os artigos e as quantidades."

A Joana, ao aceder ao rascunho com a nota, verifica que não está listado o artigo que tinha pedido e, ainda, que estava listado mais um artigo dentro de uma outra classe de artigos mais caros. Apressa-se a avisar o Gonçalo do sucedido, introduzindo no rascunho mais uma nota junto da referência a mais:

"Não vejo nenhuma caixa de chocolates de 12 e verifico que está aqui uma caixa a mais nas de 24, queira alterar sff."

Ao que o Gonçalo responde com uma nova nota:

"Joana, o artigo que refere já não se encontra disponível este ano, tomei a liberdade de enviar mais uma de 24, procedendo ao desconto do excedente. Quer que retire o artigo de qualquer forma?"

A Joana responde com uma nova nota junto ao valor final:

"Sendo assim, o valor está correto, obrigado pela diligência e simpatia. Pode emitir a fatura. Um Feliz Natal".“

3.2 Storyboard

A técnica de *storyboarding* é uma técnica usada no desenvolvimento de *software*, e que ajuda a identificar algumas das partes fundamentais da interação do utilizador com o sistema. Esta é útil para perceber os contextos em que os utilizadores irão usar o *software* e permite

explorar potenciais experiências, antes de investir no desenvolvimento do *software* propriamente dito[4]. A lógica por detrás desta técnica é, tal como em outras técnicas usadas neste capítulo, capturar mais cedo possível o maior número de detalhes e requisitos do sistema, antes mesmo de um protótipo. O *storyboard* aqui desenhado tem em conta o cenário acima descrito.

Analisando o *storyboard* apresentado (**Figura 3.1**), torna-se claro que existem três pontos distintos de contacto com o utilizador. Um dos pontos de contacto é a aplicação Windows. A implementação de um cliente *Windows Store App* era uma das premissas iniciais do projeto. Outro dos pontos de contacto é o ERP Primavera e por fim o UserSpace Primavera. Este é um portal da Primavera que tem como objetivo chegar não só a cada uma das organizações que são clientes da Primavera, mas principalmente a cada uma das pessoas que pertencem a estas, criando um ponto de encontro onde se poderão encontrar diversas informações e outros temas de interesse para as pessoas.

Neste ponto foi necessário tomar uma decisão sobre onde seriam apresentadas as notas no ERP. Por um lado, o serviço poderia ser consumido em algum controlo implementado diretamente no ERP mas, por outro lado, existia a hipótese de criar um componente WebCentral, que poderia ser usado tanto no ERP como no UserSpace Primavera. A plataforma WebCentral é uma plataforma desenvolvida pela Primavera, que permite a construção de portais *web*, a qual será detalhada mais à frente.

A vantagem da utilização do componente, é que está alojado num servidor *web* e o mesmo componente pode ser usado em dois ambientes distintos, facilitando a manutenção e reduzindo os pontos de falha. No entanto, a utilização deste componente no ERP traz uma menor flexibilidade, além de colocar mais limitações à ligação entre ambos do que a esperada num controlo nativo. Os pormenores de implementação serão detalhados mais à frente.

A escolha recaiu no componente WebCentral, dado que, o mesmo pode também ser usado de forma transparente no UserSpace, o que permite que um maior número de utilizadores possa usufruir do serviço, sem estar preso à aplicação Windows e ao ERP.

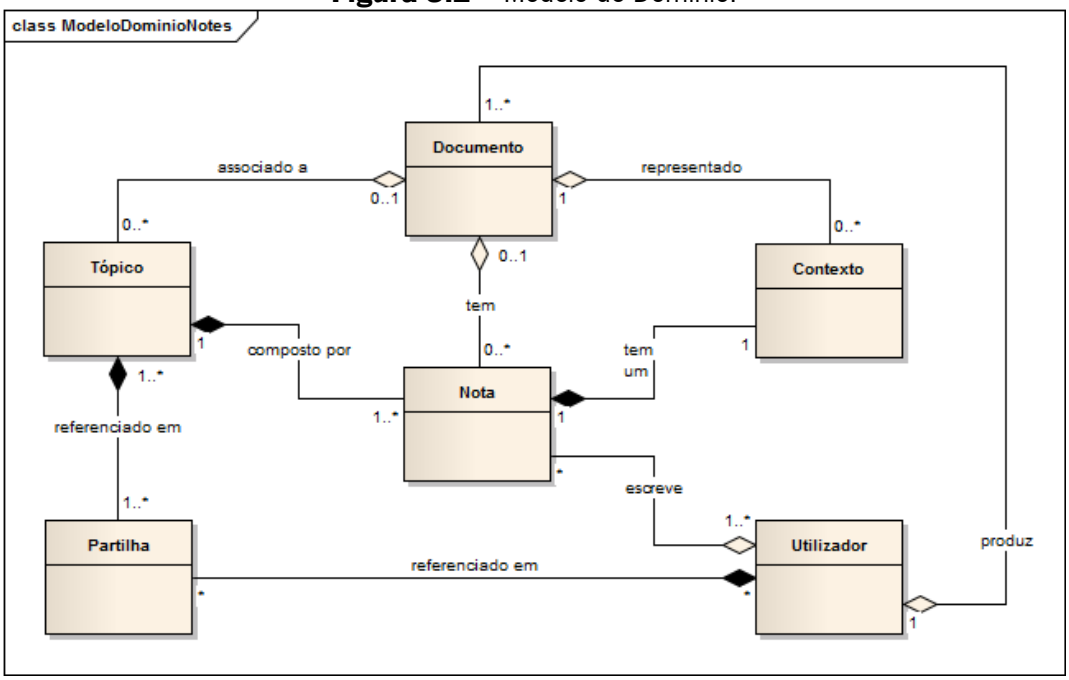
[4] http://indigo.infragistics.com/help/what-are-storyboards-good-for.html

Figura 3.1 – *Storyboard*

3.3 Modelo de Domínio

Após a análise do cenário de utilização desenhado e respetivo *storyboard*, foram definidos os conceitos fundamentais do domínio do problema, os quais estão representados no modelo do domínio (**Figura 3.2**).

Figura 3.2 – Modelo de Domínio.



Neste modelo foram também modeladas as relações entre os conceitos definidos, permitindo assim, capturar papéis e restrições que possam existir.

O conceito chave neste problema é o conceito de **Nota**, sendo que esta se encontra no epicentro dos restantes conceitos. As notas estão normalmente associadas a um **Documento**, sendo este um documento qualquer do ERP. Apesar de apresentarem uma forte relação, as notas podem continuar a existir após a destruição do documento, ou até existirem sem associação a um documento. Uma nota pode ter também um **Contexto**, que é uma representação simplificada do documento a que a nota está associada e que só existe se associado a esta. Um conjunto de notas está agrupado sob um **Tópico**, sendo que vários destes podem estar ligados a um documento. Cada tópico está referenciado em diferentes **Partilhas,** que efetuam a ligação entre **Utilizadores** e tópicos. Uma nota tem ainda uma ou mais **Definições de Utilizador**, consoante o número de utilizadores que têm acesso a ela. O **Utilizador** é um autor das notas e pode também ser o criador do documento.

3.4 User Stories

De forma a melhor capturar quais as funcionalidades necessárias no sistema, foi escrito um conjunto de *user stories,* que serão descritas de seguida. As *user stories* permitem ter uma ideia de alto nível das funcionalidades que os utilizadores esperam encontrar no sistema e do resultado esperado das suas ações. Estas seguem normalmente um formato comum:

"Como (papel), quero (alguma coisa), de forma a (obter um benefício)."

Este formato obriga a pensar em quem terá o benefício de uma certa funcionalidade e porquê. A utilização de *user stories* é uma forma de pensar nos requisitos do sistema e assim planear o que será implementado. Cada *user story* deve incluir um critério de aceitação, o qual permitirá posteriormente validar se esta foi implementada corretamente.

Tendo em conta a natureza deste projeto, devido ao facto de os serviços serem consumidos por dois clientes distintos, regra geral, o objetivo é replicar as funcionalidades nos dois, com pequenas diferenças a explicar posteriormente.

1. **Consulta de Tópicos:** como utilizador do sistema quero ter a possibilidade de consultar tópicos de notas, de forma a poder encontrar mais facilmente a nota que pretendo encontrar.

   Critério de Aceitação:

   a) O utilizador tem acesso a uma lista de tópicos partilhados consigo.
   b) Cada tópico deve ter uma nota "pai" visível ao utilizador.

2. **Criação de Tópicos:** como utilizador do sistema quero ter a possibilidade de criar tópicos de notas, de forma a poder iniciar uma conversação com outro utilizador.

   Critério de Aceitação:

   a) O utilizador pode criar um tópico associado a um documento no ERP.
   b) O utilizador pode criar um tópico sem associação noutro ambiente que não o ERP.

3. **Remoção de Tópicos:** como utilizador do sistema quero ter a possibilidade de remover tópicos, de forma a poder controlar os tópicos acessíveis.

   Critério de Aceitação:

   a) A remoção do tópico remove-o de todos os utilizadores.
   b) Apenas o utilizador criador do tópico pode removê-lo.

4. **Responder a um Tópico**: como utilizador do sistema quero ter a possibilidade de responder a um tópico partilhado comigo.

   Critério de Aceitação:

   a) Apenas é possível responder em tópicos partilhados com o utilizador.
   b) A resposta fica associada à nota "pai".
   c) Todos os utilizadores com os quais a nota pai foi partilhada conseguem ver a resposta.

5. **Partilha de Tópicos:** como utilizador do sistema quero ter a possibilidade de partilhar um tópico e respetivas notas relacionadas, para que outros utilizadores do serviço possam ter acesso a um tópico.

   Critério de Aceitação:

   a) O conjunto de todas as notas relacionadas fica acessível aos utilizadores com os quais a partilha foi feita.

6. **Pesquisa de Notas:** como utilizador do sistema quero ter a possibilidade de procurar notas através de palavras, para que possa encontrar as notas que pretendo com maior rapidez.

   Critério de Aceitação:

   a) A pesquisa é feita no conteúdo textual das notas e nos nomes dos criadores das mesmas.

7. **Filtragem de Tópicos (Temporal):** como utilizador do sistema quero ter a possibilidade de filtrar os tópicos, de acordo com um conjunto de intervalos temporais pré-definidos, de forma a poder encontrar as notas que pretendo.

   Critério de Aceitação:

   a) Os intervalos devem ser: últimos 30 dias, últimos 8 dias, ontem, hoje.
   b) A filtragem deve ter em conta a data da última nota de um tópico.

8. **Ordenação de Tópicos:** como utilizador do sistema quero ter a possibilidade de ordenar a lista de tópicos por um conjunto de critérios pré-definidos, de forma a poder encontrar mais facilmente as notas que pretendo.

   Critério de Aceitação:

   a) Os critérios devem ser: Mais Recentes primeiro, Mais Antigas primeiro, Autor A-Z, Autor Z-A e Não Lidas.

9. **Indicação de notas não lidas:** como utilizador do sistema quero ter a possibilidade de visualizar quantas e quais são as notas não lidas, de forma a poder ter uma pista visual de eventuais notas que ainda não tenha lido.

   Critério de Aceitação:

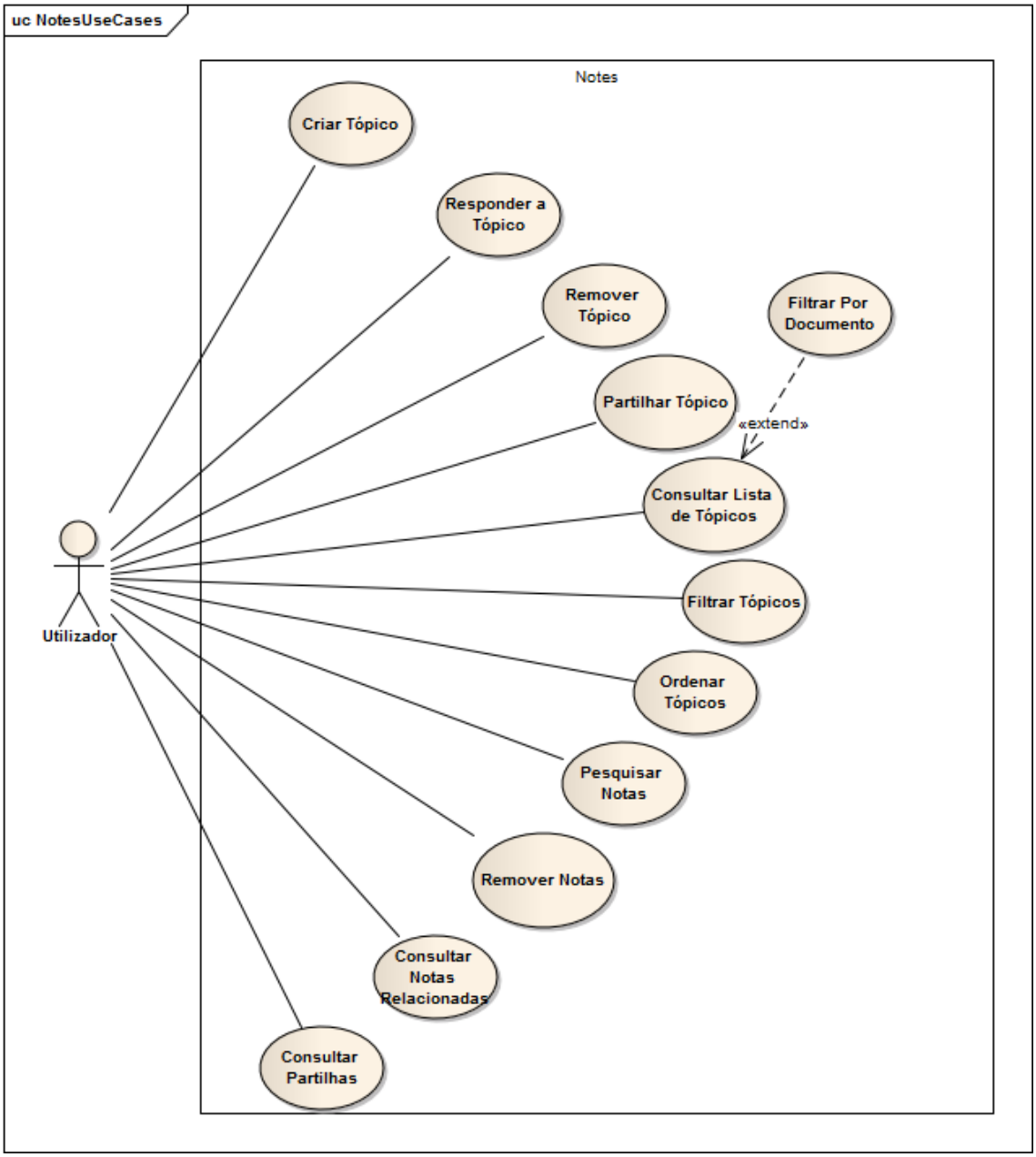
   a) Deve existir um contador de notas não lidas.
   b) As notas não lidas devem ter um estilo visual diferente das lidas.

Nesta secção foram escritas nove *user stories* e respetivos critérios de aceitação.

3.5 Casos de Uso

Após a escrita das *user stories,* foram definidos os casos de uso do sistema. Os conceitos são semelhantes, mas com algumas diferenças importantes.

Figura 3.3 – Diagrama de casos de uso



Uma *user story* reflete, num formato de alto nível, uma necessidade do utilizador, enquanto um caso de uso deve definir detalhadamente o comportamento do sistema para cumprir essa necessidade. Numa *user story*, o leitor deve ficar rapidamente com uma ideia da

necessidade que o sistema deve suprir, uma vez que são escritos numa linguagem fácil de entender. Já um caso de uso deve conter mais detalhe de forma a ser claro e não ambíguo.

Foram definidos os casos de uso, os quais estão representados na Figura 3.3. De seguida é apresentado, como exemplo, a especificação de um dos casos de uso:

Listagem 3.1 – Caso de uso: "Responder a um tópico"

Título	Responder a um tópico.		
Ator	Utilizador		
Descrição	O utilizador pretende responder a um tópico.		
Pré-Condição	O utilizador deve estar autenticado. O utilizador deve estar a visualizar um tópico já existente. O tópico deve estar partilhado com o utilizador.		
Pós-Condição	Sucesso: A nota fica guardada no sistema. Os utilizadores com os quais este tópico está partilhado devem ter acesso à nova nota. Esta nova nota deve ser visualmente diferente das restantes e deve ficar associada à nota pai do tópico.		
Fluxo Normal de Eventos		Ator	Sistema
	1	O utilizador insere o texto que pretende.	
	2	Submete a nova nota ao sistema.	
	3		O sistema verifica se a nota traz contexto associado.
	4		O sistema guarda a nota no sistema.
	5		O sistema cria uma definição de utilizador da nota criada para cada um dos utilizadores que têm acesso ao tópico.
			O sistema retorna a mensagem de sucesso.
Fluxo Alternativo 3a		Ator	Sistema
	1		O sistema deteta que a nota não traz contexto.
	2		O sistema copia o contexto da nota anterior.
	3		O fluxo continua no passo 4.

Como se pode verificar, existe uma *user story* com o mesmo título deste caso de uso. Nem sempre acontece um mapeamento de um para um, podendo uma *user story* dar origem a vários casos de uso. Tal como foi referido, o caso de uso inclui um maior detalhe da interação do utilizador com o sistema. Este permite saber claramente quais as ações principais que devem ocorrer, para que a interação seja concluída com sucesso.

Em síntese, ambos são importantes numa análise de requisitos, mas com propósitos diferentes. A *user story* pode ser mais útil numa fase inicial da modelação do sistema, enquanto um caso de uso será mais útil numa fase posterior.

3.6 Conclusões da Análise de Requisitos

Como já foi referido anteriormente, as soluções de ERP consistem, normalmente, em aplicações que integram todos os dados e processos de uma organização num único sistema, facilitando a automatização de processos. Esta organização e automatização de processos segue, normalmente, um conjunto de regras formais, imprescindíveis ao dia-a-dia de uma empresa, no entanto, é fácil e comum deixar-se de lado uma parte essencial para qualquer empresa: as pessoas, as relações entre elas, o dia-a-dia, a comunicação, as sinergias, etc. A formalidade pode ser um ponto forte e decisivo em qualquer sistema desta natureza, no entanto, existe uma lacuna nestes sistemas que normalmente tem de ser preenchida com o auxílio de outras ferramentas/soluções.

A comunicação informal entre as pessoas é um contributo essencial para uma empresa funcionar mais fluida e harmoniosamente. Normalmente é difícil encontrar espaço neste tipo de ferramentas para soluções que facilitem a comunicação entre as pessoas de uma forma útil e ligada aos processos formais.

3.6.1 Notes

A solução encontrada consiste na troca de simples notas entre utilizadores, seguindo uma lógica de conversação, no entanto, esta troca de notas pode ter associada a si um contexto de negócio, algo relacionado com algum processo da empresa, mas em que seja necessário haver algum tipo de discussão entre duas ou mais pessoas.

O que traz aos utilizadores?

A criação de uma funcionalidade de comunicação abre um novo universo de potencial de experiência que se pode revelar importante do ponto de vista do utilizador nas seguintes dimensões de necessidade:

1. Comunicação intraempresarial

   O contexto de negócio associado à conversação entre utilizadores facilita a análise e a interpretação dos documentos trocados no *workflow* interno das empresas. Normalmente existem pessoas dentro de uma empresa com maior conhecimento em certas áreas que outras. O objetivo da solução é facilitar a comunicação entre as pessoas, tornando o seu dia-a-dia mais fácil e agilizando, assim, processos.

2. Comunicação interempresarial

   Da mesma forma que a comunicação intraempresarial facilita a análise e interpretação de documentos trocados no *workflow* interno das empresas, também a abertura da comunicação a outras empresas facilitará a interpretação dos documentos trocados no *workflow* externo entre empresas.

3. Distribuição de tarefas

   Numa tarefa típica, é raro uma pessoa ter toda a informação que necessita para desempenhar o seu trabalho e gerar entidades informativas. A solução visa facilitar o preenchimento de uma entidade informativa ou documental (e.g. uma fatura)

Neste capítulo foi feito um levantamento dos requisitos necessários para o âmbito desta dissertação. Foram usadas algumas técnicas de levantamento de requisitos, como o uso de *storyboards*, *user stories,* modelo de domínio e casos de uso. Tal como foi apresentado na introdução deste documento, aquilo que se pretende é a construção de um serviço que permita comprovar que uma arquitetura orientada a serviços pode servir de base à implementação de novas funcionalidades, de uma forma mais sustentável e com a facilidade de interoperabilidade com plataformas diferentes. É nesse âmbito que surge também a necessidade de implementar uma *Windows Store App*, a qual servirá para provar o conceito.

Capítulo 4

Arquitetura

Este capítulo irá focar-se na explicação da arquitetura de todas as peças envolvidas no projeto. Não só na arquitetura pensada e desenvolvida neste projeto, mas também da *framework* de desenvolvimento de *software* Primavera sob a qual assenta a solução. Um dos objetivos deste projeto foi também fazer uso dessa *framework* no âmbito dos projetos de *software* orientados a serviços e em ambientes *cloud*. Na primeira secção é explicada a arquitetura das soluções Primavera usadas e de seguida o desenho da solução implementada no âmbito deste projeto.

4.1 Infraestrutura

Nesta primeira secção será feita uma breve explicação da infraestrutura Primavera usada como base no desenvolvimento desta dissertação. Isto permitirá contextualizar as decisões tomadas na definição da arquitetura da solução.

4.1.1 Framework Athena

No passado, o desenvolvimento de *software* era uma tarefa artesanal. Milhares de linhas de código eram escritas manualmente com índices de reutilização baixos e com alta probabilidade de se encontrarem falhas graves, até mesmo de segurança. Mesmo hoje em dia, alguns produtos de *software* enfrentam vários desafios de difícil resolução: tecnologia desatualizada, complexidade elevada nas soluções existentes, qualidade baixa e difícil manutenção, além de dificultarem a inovação.

A *framework* Athena é uma solução, desenvolvida pela Primavera BSS, já utilizada no desenvolvimento de algum *software* na empresa. Esta *framework* permite aos programadores efetuar a modelação das entidades de negócio e respetivos serviços agilizando, assim, o

processo de desenvolvimento de *software* com a automatização de vários aspetos do desenvolvimento, e com a redução dos pontos de intervenção manual do programador na conceção de um produto, também com o objetivo de reduzir o número de falhas esperado.

A *framework* Athena foi então projetada com três grandes objetivos:

- Desenhar uma *framework* que permitisse suportar o desenvolvimento da próxima geração de produtos Primavera;
- Aumentar a produtividade do desenvolvimento de *software,* através da diminuição significativa dos artefactos de *software* desenvolvidos manualmente e consequentemente, aumentar a disponibilização de mais produtos.
- Aumentar a qualidade dos produtos, com produtos mais focados nos processos de negócio e com índices de usabilidade e facilidade de adaptação de novas funcionalidades superiores. Além disso, reduzir o número de problemas.

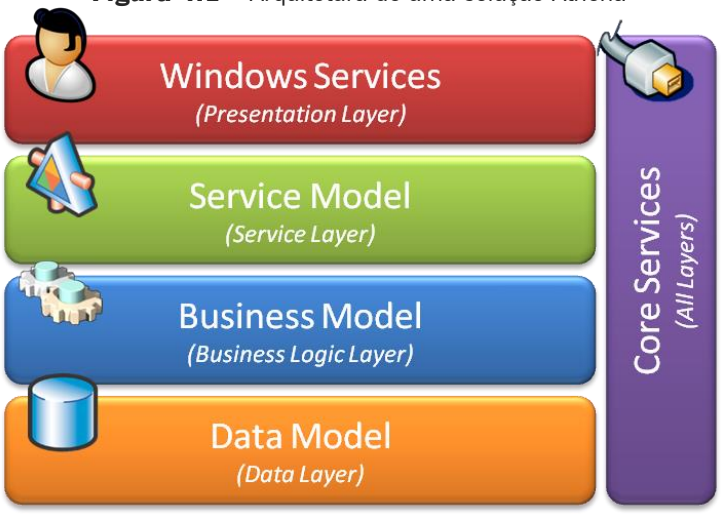
Arquitetura

A arquitetura de uma solução desenvolvida com a *framework* Athena segue um conjunto de conceitos, que são aplicados em qualquer produto desenvolvido:

- Orientação aos serviços;
- *Design Patterns (Adapter, Service Stub, Singleton, Observer, Proxy, Abstract Factory);*
- *Business Patterns;*
- Programação Declarativa;

O seguinte diagrama demonstra uma visão geral deste desenho arquitetural:

Figura 4.1 – Arquitetura de uma solução Athena



Esta arquitetura é composta por 4 camadas:

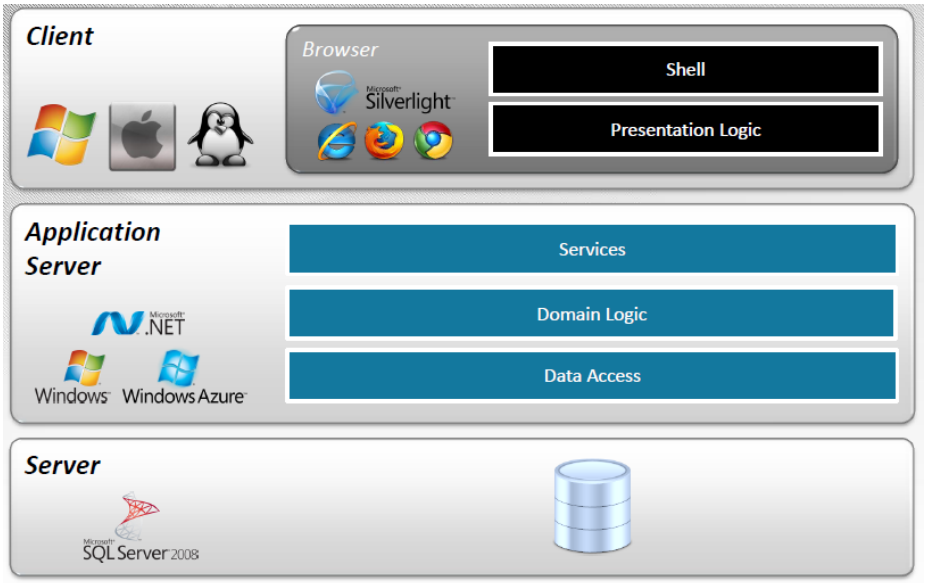
- **Camada de Apresentação**: camada de interação com o utilizador;
- **Camada de Serviços**: camada que inclui os serviços que as aplicações cliente (camada de apresentação) usam para comunicar com as camadas inferiores;
- **Camada de Negócio**: camada que trata a lógica de negócio da aplicação;
- **Camada de Dados**: camada responsável para lidar com a representação dos dados, a sua persistência e extração.

Esta arquitetura dividida em camadas está preparada para ser implementada num ambiente multinível, suportando configurações diferentes:

- **3-níveis:** cliente, *application server* e servidor;
- **2-níveis:** cliente/servidor;
- **1-nível:** todas as camadas numa única máquina.

Numa configuração de 3 níveis, a organização pode ser vista da seguinte forma:

Figura 4.2 – Arquitetura de um ambiente multinível



O cliente pode assumir várias formas. Pode ser uma aplicação *desktop* (chamadas de *smart-clients*) de qualquer sistema operativo *desktop* (Windows, MacOS, Linux), uma aplicação *web* que é executada num navegador *web,* ou ainda, uma aplicação desenvolvida para algum dispositivo móvel. Na verdade, pode ser qualquer tipo de aplicação, desde que consuma os serviços expostos ao nível do *Application Server.*

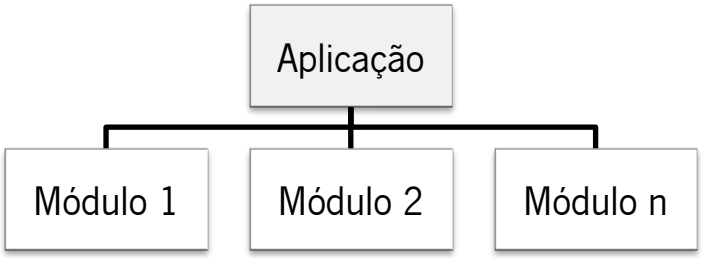
O *Application Server* corre todas as aplicações e tipicamente é um servidor como o Microsoft Internet Information Services (IIS). O *Server* na verdade é um servidor de base de dados que aloja o sistema de gestão da base de dados (SQL Server).

Desenvolvimento usando a framework Athena

Para desenvolver um produto Athena é necessário um SDK[5], desenvolvido pela Primavera, e que integra com a ferramenta *Microsoft Visual Studio*. Em conjunto com este SDK é necessário também instalar algumas extensões ao *Visual Studio*. Os produtos desenvolvidos com a *framework* são modelados com recurso a um conjunto de *designers*. Estes *designers* são extensões ao *Visual Studio,* que permitem fazer a modelação graficamente.

Os produtos são compostos por um componente chamado Aplicação (neste documento será também usado o termo Produto ou *Application*) e por um ou mais componentes chamados Módulos.

Figura 4.3 – Estrutura de um produto Athena



Aplicação

O componente Aplicação (**Figura 4.3**) é o componente que agrega os diversos módulos de uma aplicação. Cada produto/aplicação desenvolvido com a *framework* irá partir sempre de um componente Aplicação.

Módulo

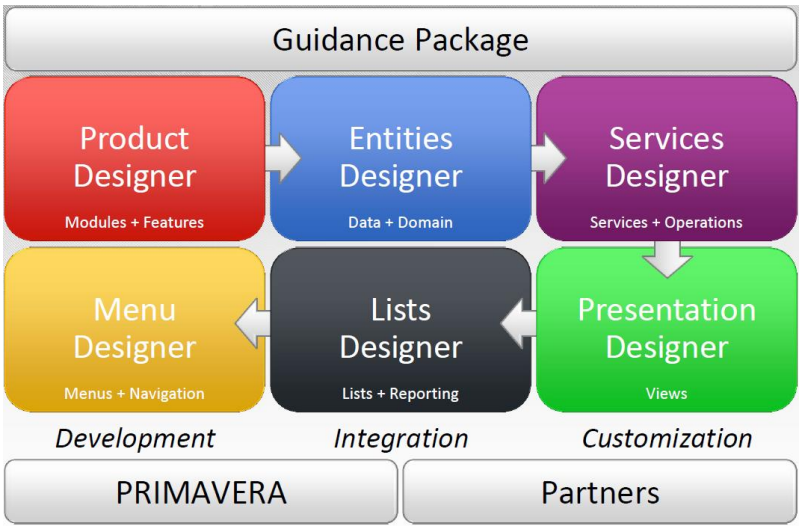
Os módulos (**Figura 4.3**) são projetos que representam componentes lógicos de um produto. As aplicações Athena podem ser compostas por diversos módulos, e os diferentes módulos podem integrar aplicações diferentes. Isto permite, por um lado, ter grande flexibilidade na construção das aplicações e, por outro, que se formem equipas de desenvolvimento especializadas numa certa área de negócio.

[5] Software Development Kit

Designers - Framework de Modelação

O ponto de partida para a criação de qualquer produto da *framework* Athena é a modelação. Como se pode ver na **Figura 4.4**, para modelar um produto Athena é necessário percorrer um conjunto de *designers*.

Figura 4.4 – Diagrama da framework de modelação



Na **Figura 4.4** pode ver-se também uma peça chamada *Guidance Package*. Esta peça consiste numa extensão ao *Visual Studio,* que inclui *templates* de projetos e um conjunto de receitas para construir Aplicações e Módulos Athena. Isto permite acelerar o arranque da construção de uma Aplicação Athena, assim como ter algumas receitas para acelerar o desenvolvimento.

Além deste *Guidance Package* temos então os *designers*:

1. ***Product Designer***: define os módulos da aplicação. Neste *designer* podem ser definidas as funcionalidades de cada módulo. Este *designer* gera o ficheiro de configuração do produto.
2. ***Entities Designer***: define o modelo de dados (domínio) das entidades de negócio. É a partir deste modelo que são geradas classes de negócio (C#) assim como os mapeamentos ORM (Entity Framework) e respetiva base de dados.
3. ***Services Designer***: define os serviços do domínio e as operações disponíveis para manipular as entidades de negócio. A partir deste modelo são gerados vários artefactos: operações CRUD[6], serviços WCF, serviços RIA e o esqueleto do

[6] Create, Read, Update, Delete.

código necessário para implementar alguma operação específica de serviço modelada.

4. ***Presentation Designer:*** define as vistas usadas para cada operação dos serviços. Gera o layout das vistas (Silverlight/XAML) e a sua lógica.
5. ***Lists Designer:*** define o modelo de dados usado para *reporting* utilizando listas *Query Builder.*
6. ***Menu Designer:*** define o menu da aplicação.

Nesta secção foram apresentadas as características da framework Athena assim como uma abordagem ao desenvolvimento de *software* recorrendo a esta.

4.1.2 Primavera Elevation

O conjunto de produtos Primavera, com forte ligação a ambientes *web/cloud,* está sob a designação de Primavera Elevation. Nesta secção será feita uma breve descrição de duas das soluções desta oferta, uma vez que são peças fundamentais na conceção do projeto implementado com esta dissertação.

CloudServices

Um dos produtos *Athena* desenvolvido pela Primavera é o produto CloudServices. Como o próprio nome indicia, este produto foi desenvolvido com o objetivo de fornecer uma plataforma de desenvolvimento de soluções, pensadas e entregues como serviços *cloud,* aos clientes Primavera. Esta plataforma inclui uma série de características associadas a este tipo de desenvolvimento, nomeadamente os conceitos de subscrição, estatísticas de utilização, faturação, administração de utilizadores, permissões, assim como outras opções de configuração.

O módulo do serviço a desenvolver neste projeto é um módulo Athena a integrar com este produto.

WebCentral

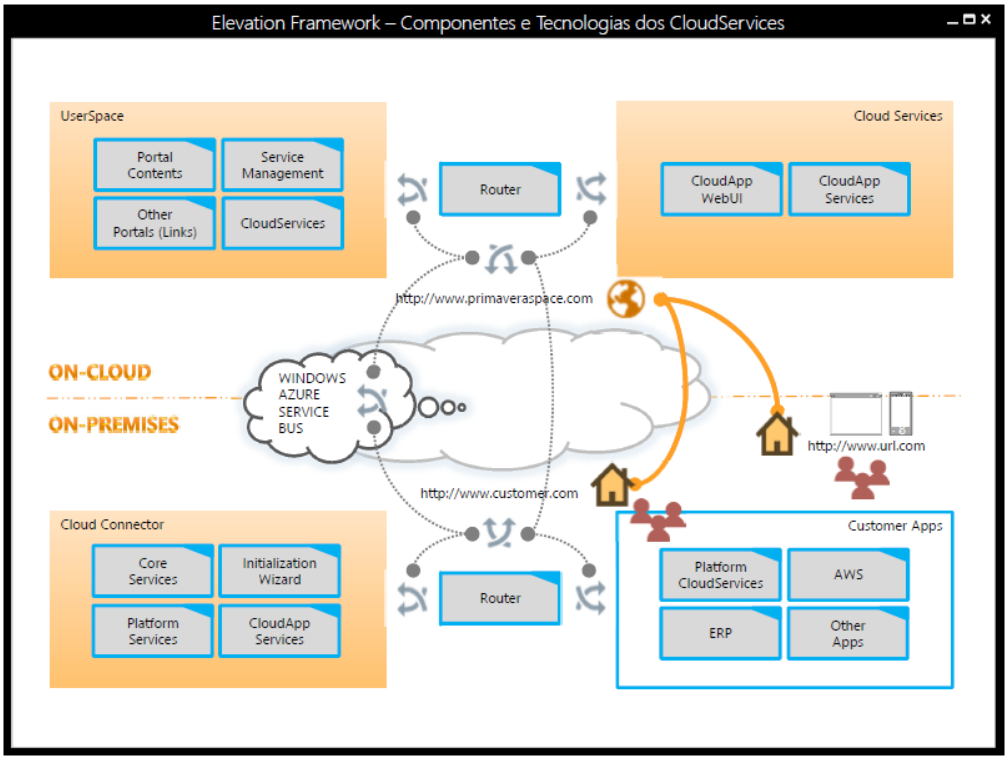
A plataforma WebCentral é uma plataforma desenvolvida pela Primavera, que permite a construção de portais web de diversos tipos, à medida das exigências de cada organização. Isto inclui tanto portais privados de empresa, como portais para o público em geral.

É sobre esta plataforma que se encontra construído o Primavera UserSpace. Como já foi anteriormente referido, este é um portal da Primavera que tem como objetivo chegar, não só a

cada uma das organizações que são clientes da Primavera, mas principalmente a cada uma das pessoas que pertencem a estas, criando um ponto de encontro onde se poderão encontrar diversas informações e outros temas de interesse para as pessoas.

É também no UserSpace que os utilizadores podem subscrever os CloudServices Primavera e tratar de toda a gestão relacionada com os serviços subscritos. Alguns dos serviços podem também ser usufruídos no próprio portal.

Figura 4.5 – Arquitetura Soluções Elevation



Retirado da Documentação Primavera Elevation Framework

Na **Figura 4.5** está representado o diagrama da arquitetura das soluções Elevation. Nesta figura estão representadas duas áreas distintas: *on-cloud* (ambiente na nuvem) e *on-premises* (ambientes locais).

Na área *on-premises* estão as peças da arquitetura que se encontram em ambientes locais. Isto inclui o ERP Primavera e quaisquer outras aplicações cliente que consumam os *CloudServices.* Nesta área encontra-se ainda outra peça, o *Cloud Connector*. O *Cloud Connector* é uma peça que facilita a comunicação entre o ERP e os *CloudServices*, integrando um conjunto de funcionalidades necessárias para essa comunicação.

Na área *on-cloud* pode ver-se as duas peças já abordadas nesta secção: *UserSpace* e *CloudServices.*

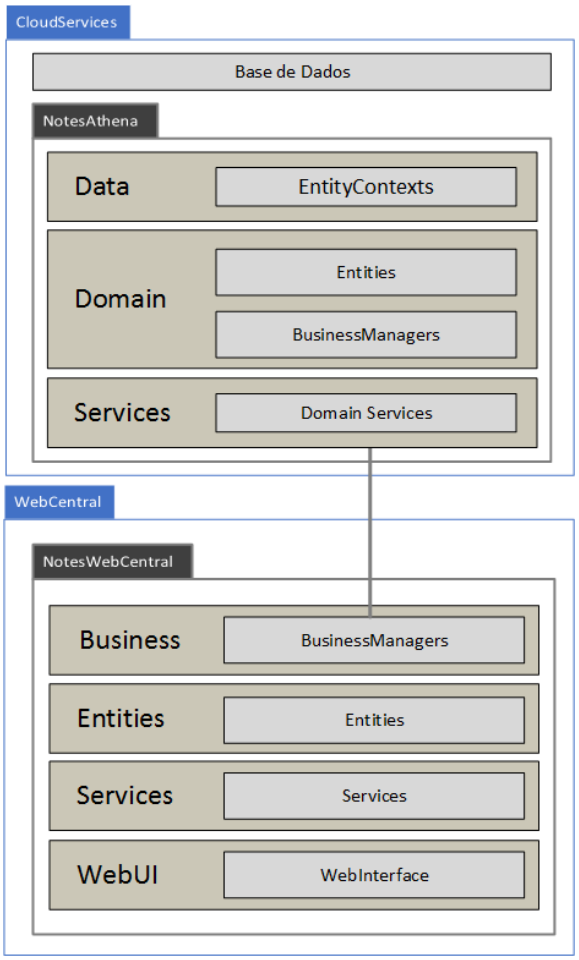
Pelo meio encontram-se os *Routers,* através dos quais se faz a comunicação, dado que são eles que expõem os *endpoints* de acesso.

4.2 Arquitetura Implementada

Após a análise dos requisitos levantados, foi então desenhada a arquitetura da solução a implementar. Na **Figura 4.6** podemos ver o diagrama de alto nível da arquitetura da infraestrutura de serviços da solução.

Nesta figura encontram-se dois módulos distintos. Na parte superior pode ver-se representado o módulo CloudServices, denominado NotesAthena. Este módulo é o módulo Athena, desenvolvido para integrar na solução Athena Primavera CloudServices. Na parte inferior encontra-se o módulo WebCentral, denominado NotesWebCentral. Este é o módulo que expõe os serviços para poderem ser consumidos em aplicações cliente, como a aplicação Windows a desenvolver.

Figura 4.6 – Arquitetura da infraestrutura dos serviços desenvolvida



4.2.1 Arquitetura do Módulo NotesAthena

A base onde toda arquitetura da solução assenta é no produto Primavera CloudServices. Tal como foi explicado anteriormente (ver 0), este é um produto construído usando a *framework* Athena.

O módulo NotesAthena é o módulo que foi desenvolvido no âmbito deste projeto e que pertence ao produto CloudServices. Neste módulo foi feita toda a modelação Athena e implementação dos serviços.

Na **Figura 4.6** podem ver-se os componentes principais deste módulo. Tal como já foi apresentado anteriormente, um módulo Athena divide-se em vários componentes, que também se encontram divididos por camadas. A base de dados é partilhada por todos os módulos de um produto, ou seja, como se pode observar no diagrama, não pertence exclusivamente ao módulo. A comunicação do módulo com a base de dados fica a cargo do componente *Data* e respetivos *EntityContexts*, classes *EntityFramework* geradas pela *framework* Athena.

No componente *Domain,* encontram-se as entidades de negócio *(Entities),* também conhecidas como POCO *(Plain Old CLR Object)*. Estas são classes simples, com as propriedades modeladas no *designer* de modelação Athena para cada entidade. Também estas classes são geradas automaticamente. Ainda neste componente, podem encontrar-se os *BusinessManagers*. Estas classes são as principais responsáveis pela lógica de negócio do módulo. A *framework* gera o esqueleto dos *managers* para as entidades modeladas, assim como métodos CRUD *(Create, Read, Update, Delete)* para cada uma delas. É, no entanto, ao nível dos *managers,* que é feita a principal intervenção ao nível da introdução de código não gerado automaticamente. É a este nível que é feita a implementação da lógica de negócio que não é abrangida pelo típico CRUD.

Por fim, o componente *Services* inclui os *DomainServices.* Estas classes são responsáveis por comunicar com os *managers* da camada inferior e expor os serviços deste módulo. Estes serviços ficam expostos em *endpoints* WCF que recorrem a mensagens SOAP como forma de comunicação.

4.2.2 Arquitetura do Módulo NotesWebCentral

Os serviços expostos pelo módulo NotesAthena não são consumidos diretamente pelas aplicações cliente, mas sim pelo módulo NotesWebCentral. Este não é gerado pela *framework* Athena, mas obedece a uma organização por camadas. Como se vê na **Figura 4.6**, a primeira

camada é a camada *Business*. Nesta encontram-se os *BusinessManagers,* responsáveis por comunicar com os serviços expostos pelo módulo NotesAthena. Estes tratam também da autenticação com o módulo NotesAthena e de traduzir os objetos devolvidos *(DataContracts)* pelo serviço, para um objeto com os dados necessários à camada de apresentação, ou seja, convertidos para um DTO *(Data Transfer Object).* No componente *Entities* encontram-se definidas as classes com a estrutura dos *DTO* necessários.

O componente *Services* inclui as classes necessárias que comunicam com os *managers* e expõem os serviços usando *webHttpBinding,* que devolvem objetos JSON. É neste módulo que também se encontra alojado o controlo web que é apresentado no ERP e no UserSpace.

Este módulo NotesWebCentral é um módulo intermédio, que consome os serviços NotesAthena e expõe novos serviços WCF, mas cuja comunicação é feita através de pedidos http (webHttpBinding) em vez de mensagens SOAP (seguindo uma abordagem REST) e cujos dados são transmitidos através de objetos JSON.

A conjugação de uma arquitetura REST com JSON tem-se tornado popular devido a um conjunto de razões. O formato JSON é um formato para troca de dados, leve e desenhado para poder ser também facilmente entendido por humanos. A sua versatilidade tem-no tornado popular face a outros formatos usados no passado, nomeadamente o XML.

A utilização desta camada intermédia no WebCentral pode levantar algumas dúvidas quanto à robustez da solução. Teoricamente, o módulo Athena deveria expor todos os serviços preparados para serem consumidos diretamente pelas aplicações cliente, no entanto, existiram algumas razões para a utilização deste método. A razão mais forte para esta decisão prende-se com a forma como a infraestrutura Primavera Elevation se encontra implementada. Os utilizadores do UserSpace (que são os utilizadores que podem ter autorização para utilizar o serviço desenvolvido) encontram-se assentes na plataforma WebCentral, tornando-se a utilização deste módulo mais vantajosa. O módulo WebCentral tem acesso a métodos de plataforma para autenticação e obtenção de informações sobre os utilizadores. Por outro lado, a exposição de serviços usando *webHttpBinding,* através de objetos JSON, permite que tanto a Windows Store App como o componente web consumam os mesmos serviços, e abre portas para outro tipo de clientes, como as aplicações para sistemas operativos de dispositivos móveis (Google Android, Apple iOS, Windows Phone, etc) que podem facilmente consumir dados JSON.

4.2.3 Aplicações Cliente

Foram implementados dois clientes: a Windows Store App, que pode ser instalada em qualquer máquina com Windows 8/RT, e um controlo web do WebCentral. Ambos consomem os mesmos serviços expostos pelo módulo NotesWebCentral.

Relativamente ao controlo web, este funciona como uma página web, mas trata-se apenas de uma parte e não de uma página inteira, daí a designação controlo. Como foi referido, este está integrado no módulo NotesWebCentral, estando assim acessível, como se de uma página web se tratasse. Um módulo WebCentral é um módulo que pode incluir camadas de apresentação ao utilizador, as quais depois podem ser usadas na construção de portais web. O UserSpace é um portal construído com esta tecnologia e onde se encontra alojado este módulo. Isto permite que um módulo alojado no mesmo sítio possa ser aplicado em diferentes locais de um portal, pelo que o mesmo controlo pode ser usado em dois sítios de forma transparente: no portal UserSpace, e no ERP Primavera, através de um controlo do ERP que simula o funcionamento de um navegador de internet e que mostra o controlo sem a página web do UserSpace.

A arquitetura da aplicação Windows seguiu o padrão arquitetural MVVM *(Model-View-ViewModel)* que será explicada de seguida.

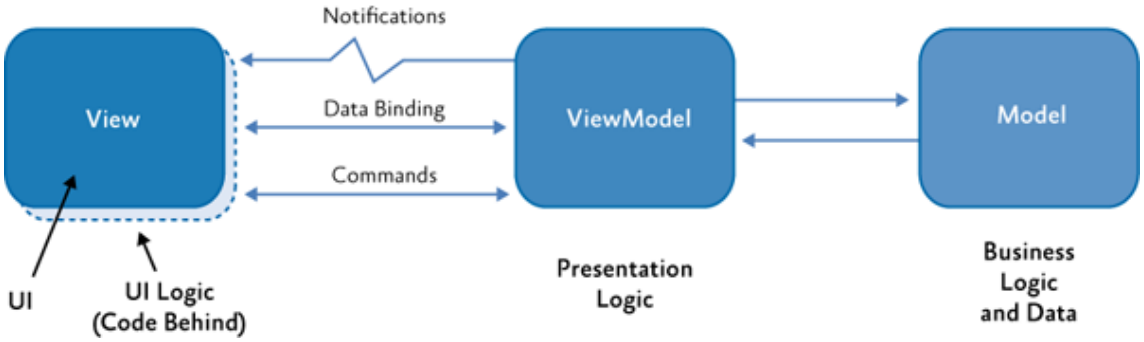
Padrão MVVM (Model – View – ViewModel)

O desenvolvimento de Windows Store Apps não obriga à utilização de nenhum padrão arquitetural. Este tipo de aplicações, tradicionalmente, segue um paradigma de programação orientada a eventos, em que o fluxo da aplicação é determinado por eventos como cliques do rato ou alguma tecla premida no teclado. Este paradigma tem, no entanto, alguns problemas, sendo que o mais notório é o código demasiado dependente da camada de apresentação.

O padrão MVVM (Model – View – ViewModel) é um padrão arquitetural originalmente desenvolvido para as aplicações da Microsoft, criadas com a tecnologia WPF e, posteriormente, também para a tecnologia Silverlight. Este padrão é uma especialização do padrão *Presentation Model*[7], criado por Martin Fowler, um conhecido autor da área do desenvolvimento de *software*.

[7] http://martinfowler.com/eaaDev/PresentationModel.html

Figura 4.7 – Diagrama de uma arquitetura MVVM



Fonte: Brumfield et al. (2011)

O MVVM (**Figura 4.7**) é um padrão semelhante ao padrão MVC (Model – View – Controller), que é um outro padrão arquitetural usado no desenvolvimento de *software*. As diferenças encontram-se no papel de cada uma das peças que compõem as arquiteturas. O *Model* e a *View* têm papéis semelhantes em ambas as arquiteturas. O *Model* representa a estrutura lógica de dados da aplicação e as classes de alto-nível associadas a ela, sendo ainda o responsável por gerir o comportamento e os dados do domínio da aplicação e definir o que é, por exemplo, um Cliente ou um Documento. Responde a pedidos de informação sobre o seu estado e a instruções para alterá-lo. O *Model* tipicamente não inclui informações sobre a camada da interface da aplicação. A *View* é responsável pela camada visual da aplicação em ambas as arquiteturas e define a estrutura e o aspeto do que o utilizador vê no ecrã. Numa *Windows Store App* esta é composta por um ficheiro de XAML (eXtensible Application Markup Language, linguagem baseada em XML), que define a estrutura e o aspeto da *View.* Este ficheiro é acompanhado de um ficheiro com o respetivo código associado, onde são programados os eventos da *View.* No entanto, numa arquitetura MVVM, este ficheiro de código deve conter apenas o estritamente necessário para o correto funcionamento da *View*.

As principais diferenças centram-se no *Controller* e no *ViewModel,* que têm papéis distintos. No MVC pode existir um controlador para várias *Views*, o qual controla os diversos eventos que ocorrem numa aplicação, informando o *Model* e a *View* para mudarem de acordo com os eventos despoletados. Já o *ViewModel* é uma abstração da *View,* que funciona como mediador entre *View* e *Model,* fazendo *queries,* conversões, validações e agregações de dados, desde que necessários pela *View*, e sem qualquer referência direta a esta. O *ViewModel* referencia diretamente o *Model* e invoca métodos deste para obter objetos do domínio, tipicamente expostos pelo *ViewModel* através de um conjunto de propriedades, posteriormente

necessárias para a construção da *View.* Além de expor o *Model,* pode manipulá-lo de forma a ser mais facilmente consumido pela *View,* fazendo, por exemplo, a concatenação de dois campos, ou ainda, definindo estados de forma a permitir que a *View* possa adaptar-se a estados diferentes. Um exemplo comum é o caso da chamada a um *web-service*, em que a *View* pode mostrar uma animação enquanto aguarda pela resposta do serviço.

A *View* liga-se ao *ViewModel* através de um mecanismo de *data-binding.* Este mecanismo permite ligar propriedades de diferentes objetos e mantê-las em sincronia, o que o torna num mecanismo muito poderoso. Isto permite ao programador focar-se em obter os valores das propriedades dos objetos necessários à *View,* sem ter de se preocupar em atualizá-la, permitindo, de igual modo, a remoção de quase todo o código necessário para manipular a *View (code-behind)*. Este é um ponto-chave na diferença entre um *Controller* e um *ViewModel. O ViewModel* mantém um estado que é apresentado pela *View,* mas não manipula os elementos da *View,* enquanto o *Controller* manipula diretamente a *View*. É através do mecanismo de *data-binding* e de um mecanismo de notificações, que existe comunicação entre as duas camadas. O utilizador interage com a *View*, que faz com que o *ViewModel* seja notificado das alterações das suas propriedades. A decisão de fazer algo com essas alterações é então da responsabilidade do programador que pode, ou não, querer alterar o *Model* ou executar qualquer outra operação. Da mesma forma, no sentido contrário, quando o *ViewModel* é alterado, as alterações são propagadas para a *View.* O *ViewModel* define também comandos/ações que podem ser representados na interface e que o utilizador pode invocar. Um exemplo comum é quando o *ViewModel* providencia um comando para submeter os dados de um formulário. A *View* pode representar esse comando com um botão, para que o utilizador possa clicar no botão para submeter os dados.

Em resumo, a principal vantagem da utilização do padrão MVVM é que promove o princípio da **separação de responsabilidades** e o princípio da **responsabilidade única** entre *Views* e *ViewModels,* uma característica sempre desejável no desenvolvimento de *software*, e de onde derivam as seguintes vantagens:

- ***Testing:*** os testes automáticos podem agora ser programados mais facilmente, cobrindo a maior parte da funcionalidade ao testar o *ViewModel.*
- **Flexibilidade:** *Views* diferentes podem ser usadas com o mesmo *ViewModel*, permitindo representações visuais diferentes da mesma funcionalidade.

- **Reutilização:** devido à separação de aspetos visuais e aspetos funcionais, tanto as *Views* como os *ViewModels* têm uma maior probabilidade de serem reutilizados do que quando têm a responsabilidade misturada.
- **Separação da programação do *design* de interfaces:** o programador pode desenvolver uma aplicação sem grandes preocupações a nível de interface enquanto o *designer* pode alterar a interface sem tocar no código da funcionalidade.

Apesar das vantagens enunciadas, a utilização do padrão MVVM pode ter também algumas desvantagens:

- ***Debug* do *Data-Binding:*** é difícil fazer *debug* ao mecanismo de *data-binding.* Uma vez que não existem referências diretas entre *View* e *ViewModel,* por vezes é difícil descobrir o porquê de uma propriedade não estar a aparecer na interface com o utilizador.
- ***ViewModel* com excesso de responsabilidade:** apesar de promover o contrário, continua a ser fácil o *ViewModel* tornar-se numa classe com excesso de responsabilidade. O facto de se usar o padrão MVVM não impede que continuem a existir más práticas de programação. Continua a ser responsabilidade do programador procurar a melhor forma de implementar as funcionalidades.
- **Falta de padrões:** apesar do padrão ser originário da Microsoft, continua a não existir um padrão na *framework* .NET. Isto faz com que exista uma proliferação de *toolkits,* desenvolvidos por terceiros, que visam facilitar o desenvolvimento de soluções usando MVVM, ao providenciar bibliotecas para o chamado *"plumbing code",* igual em todas as soluções. A falta de bibliotecas nativas da *framework* faz com que o código possa ficar dependente destas *toolkits*.

Após pesar as vantagens e desvantagens da utilização desta arquitetura, decidiu-se apostar na implementação seguindo este padrão, principalmente devido à sua promoção da separação de responsabilidades e ao facto de facilitar os testes automáticos.

Capítulo 5

Desenvolvimento do Caso de Estudo

A solução concretizada no âmbito desta dissertação foi desenvolvida com recurso a diferentes tecnologias e ferramentas. Estas podem ser, Por um lado, tecnologias de conhecimento geral, como a linguagem de programação orientada a objetos da Microsoft: o C#; a linguagem de programação interpretada por clientes como navegadores de internet: *Javascript*, e padrões como o HTML e, por outro lado, tecnologias desenvolvidas pela própria empresa Primavera, como a framework Athena.

Neste capítulo serão abordados de forma detalhada alguns aspetos da implementação. Serão ainda mostrados alguns excertos de código relevantes e imagens das aplicações implementadas.

5.1 Implementação do Módulo NotesAthena

Nesta secção serão explicadas as decisões tomadas na implementação do módulo Notes do produto CloudServices. Nesse módulo foi feita a modelação das entidades de negócio necessárias para suportar a solução.

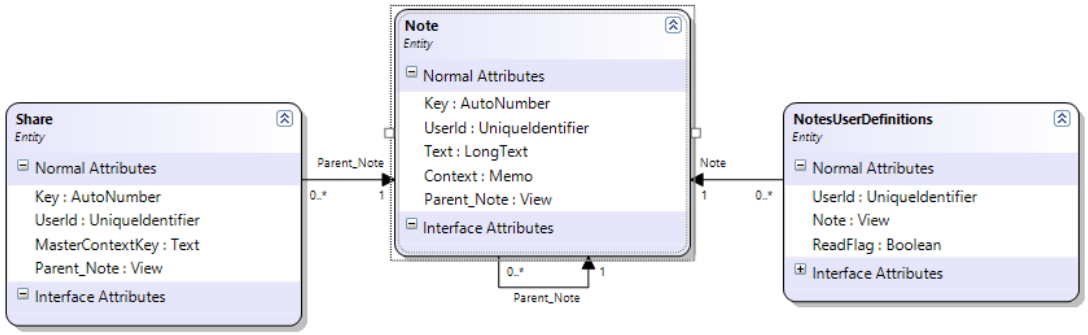
Como foi demonstrado em 0, o processo de modelação em Athena é composto por um conjunto de passos. Cada um destes consiste na definição de um modelo usando um dos *designers* definidos. Tendo em conta, no entanto, que este projeto consiste num módulo Athena para o produto CloudServices, o primeiro *designer* não é usado*: product designer,* uma vez que este é apenas usado na definição de um novo produto. Outros *designers* ficaram também fora desta modelação: *presentation, lists e menu designer*, uma vez que são mais focados em aspetos necessários na interface *Silverlight* gerada pela *framework*, a qual não é usada no âmbito desta dissertação.

5.1.1 Entidades NotesAthena

O primeiro modelo construído foi o modelo de entidades (**Figura 5.1**). Este define o modelo de dados (domínio) das entidades de negócio. É a partir deste modelo que são geradas classes de negócio (C#) assim como os mapeamentos ORM (Entity Framework) e respetiva base de dados.

Cada uma das entidades modeladas herda automaticamente de uma entidade base, um conjunto de propriedades transversais a todas as entidades de negócio. Estas propriedades incluem datas de criação, modificação e os autores das mesmas, assim como algumas *flags* de controlo.

Figura 5.1 – Modelo de Entidades NotesAthena



Foram então definidas três entidades: **Note**, **NotesUserDefinitons** e **Share**. Na *framework* Athena não é necessário modelar a entidade que representa um utilizador, uma vez que esta é "oferecida" pela própria *framework.* A entidade **Note** é a entidade base de toda solução. Esta inclui as propriedades necessárias para cada uma das notas escritas. Na **Figura 5.1**, podem ver-se as seguintes propriedades:

- **Key:** a chave natural de cada uma das notas;
- **UserId:** identificador único do utilizador que escreveu a nota. Esta é a chave estrangeira que identifica o utilizador.
- **Text:** o texto da nota;
- **Context:** serialização simplificada do contexto (documento) em que a nota foi escrita. (e.g.: fatura)
- **Parent_Note:** identificador da nota pai. Permite saber se a nota tem uma nota pai relacionada.

Relativamente à entidade **NotesUserDefinitions**, esta funciona como entidade de ligação entre os utilizadores e as notas. É um dos dois relacionamentos entre notas e utilizadores (o outro é o relacionamento de um utilizador para muitas notas - 1:N). Neste caso, esta entidade permite ter definições de cada nota para cada utilizador, apesar de, por enquanto, apenas ter uma propriedade que indica se a nota foi lida ou não. A chave desta entidade é composta pelo UserId e pelo identificador da Nota. As propriedades desta entidade são:

- **UserId:** identificador do utilizador a que pertencem estas definições;
- **Note:** identificador da nota a que são aplicadas estas definições;
- **ReadFlag:** *flag* que indica se um utilizador já leu a nota ou não.

Por fim, temos a entidade **Share.** Esta é responsável pelos dados das partilhas feitas. Ela guarda o identificador da nota pai e o utilizador com o qual a partilha foi feita. A partilha é sempre feita pela nota pai. As propriedades desta entidade são:

- **Key:** chave natural da partilha;
- **UserId:** identificador do utilizador com o qual a partilha foi feita;
- **MasterContextKey:** chave criada que permite, dentro do ERP, apresentar apenas a lista de notas relacionadas com o contexto aberto.
- **Parent_Note:** identificador da nota pai.

5.1.2 Serviços NotesAthena

Tendo em conta a temática deste projeto, a camada de serviços desenvolvida inclui essencialmente as operações de consulta, criação e remoção de notas, que são serviços com um nível de complexidade baixo. Nesta secção são apresentados os serviços desenvolvidos, explicitando a funcionalidade de cada um.

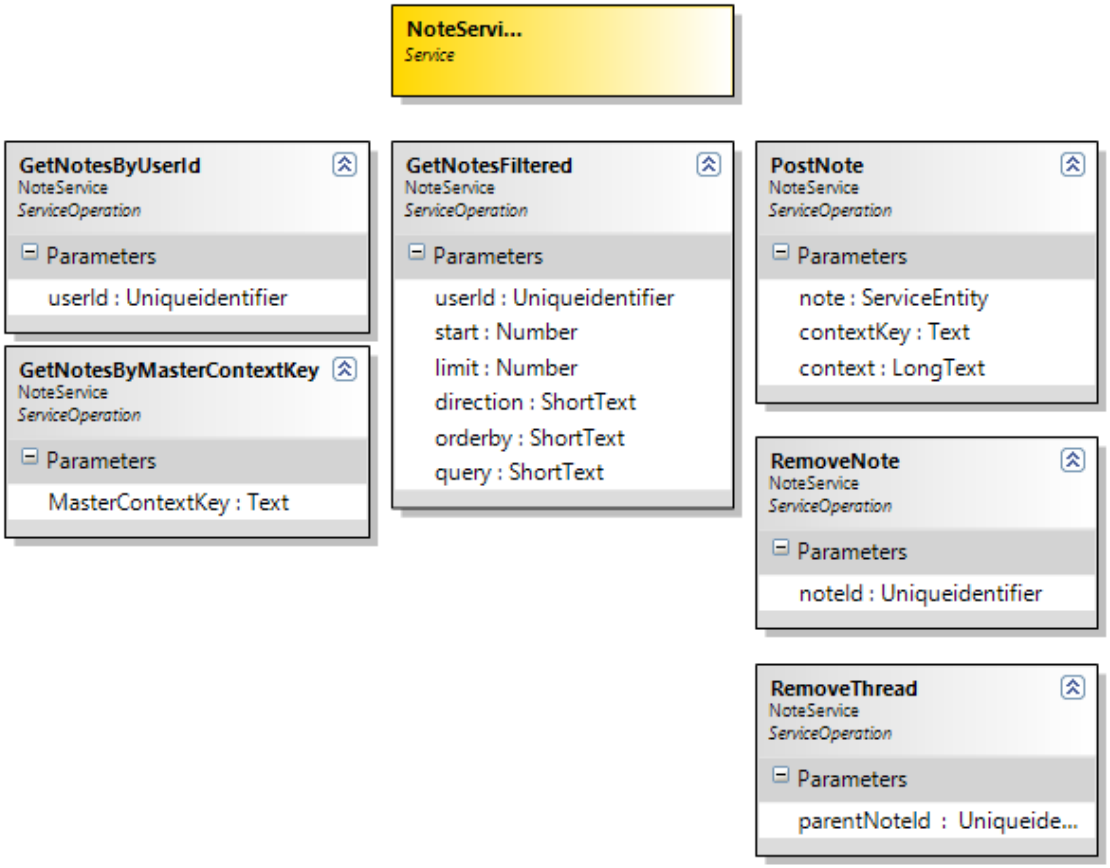
Um dos princípios da arquitetura orientada a serviços é a definição de serviços mais próximos dos processos de negócio. A operação de consulta de notas é uma operação que pode tomar várias formas, pelo que foram criadas suboperações orientadas a cada uma das formas de consulta necessárias. A criação destas operações (a modelação efetuada está demonstrada na **Figura 5.2**) acaba também por torná-las mais eficientes (do ponto de vista do tráfego de rede, por exemplo) e permite uma redução da lógica de negócio necessária do lado das aplicações cliente.

Foram então criadas as seguintes operações:

- **GetNotesByUserId:** esta operação permite obter todas as notas relacionadas com o utilizador que efetua o pedido. Isto inclui não só as notas criadas pelo utilizador, mas também todas as notas partilhadas com o mesmo.
- **GetNotesByMasterContextKey:** operação que permite obter as notas relacionadas com a janela ativa no ERP. Um utilizador que esteja a trabalhar sobre alguma entidade do ERP, com acesso à funcionalidade de notas, deve visualizar apenas as notas relacionadas com essa entidade. Esta operação permite obter apenas as notas dessa entidade.
- **GetNotesFiltered:** operação que permite filtrar as notas por um conjunto de termos de pesquisa.
- **PostNote:** criação customizada de uma nota obedecendo a algumas regras de negócio.
- **RemoveNote:** remoção customizada de uma nota obedecendo a algumas regras de negócio.
- **RemoveThread:** remoção de um conjunto de notas. Apesar do conceito *Thread* não existir no módulo Athena, optou-se pela implementação deste método a um nível mais baixo e obedecendo a um conjunto de regras de negócio.

Apesar da *framework* Athena "oferecer" operações de criação e remoção básicas, surgiu a necessidade de implementar alguma lógica de negócio extraordinária à lógica de base, de forma a garantir a correta implementação da funcionalidade.

Figura 5.2 – Modelo de serviços NotesAthena



Na **Figura 5.2** pode ver-se a modelação efetuada para os serviços Athena. Esta modelação, devidamente transformada pela *framework*, vai criar nos *Managers* a estrutura necessária para o programador apenas necessitar de introduzir a lógica de negócio que pretenda.

A título de exemplo pode ver-se de seguida a implementação do método *GetNotesByUserId:*

Listagem 5.1 – Método C# GetNotesByUserId da classe NoteManager

```
/// <summary>
/// Gets all the Notes which the user has access to.
/// </summary>
/// <param name="userId">The user id of the callee.</param>
/// <returns>The list of notes.</returns>
public override IQueryable<Note> GetNotesByUserId(Guid userId)
{
    ShareManager shareManager = new ShareManager();

    // Gets the shares for the user
    var userShares = shareManager.GetShares()
                        .Where(s => s.User == userId)
                        .Where(s => s.IsDeleted == false)
                        .ToList();

    // Gets the notes which have a parent note with the same id as the one on the share
    // and the notes that the user is owner of.
    var notes = (from share in userShares
                 join note  in this.GetNotes()
                                on share.NoteId equals note.Parent_NoteId
                 select note)
                   .Union(
                 from share in userShares
                  join note  in this.GetNotes()
                             on share.NoteId equals note.Id
                  select note)
                   .Union(
                 from note in this.GetNotes()
                  where note.Owner == userId
                  select note);

    IQueryable<Note> nts = new List<Note>(notes).AsQueryable();

    return nts;
}
```

Tal como foi já referido anteriormente, este método retorna todas as notas a que um utilizador tem acesso. No excerto de código acima, numa primeira fase, é feita uma *query* ao Manager que gere as partilhas *(ShareManager), a partir* do qual se obtêm as partilhas do utilizador. De seguida é feita uma *query* que obtém as notas que o utilizador tem acesso, tendo em conta as partilhas.

5.2 Implementação do Módulo NotesWebCentral

O módulo NotesWebCentral é um módulo intermédio entre o módulo NotesAthena e as aplicações cliente (ver 4.2). Este módulo consome os serviços expostos pelo módulo NotesAthena e expõe novos serviços WCF, sendo a comunicação feita através de pedidos http (webHttpBinding) em vez de mensagens SOAP (seguindo uma abordagem REST), e os dados transmitidos através de objetos JSON. Este módulo foi desenvolvido de forma a tentar mapear, da melhor forma possível, os serviços expostos pelo módulo NotesAthena, para assim existirem poucas discrepâncias entre os módulos, mas seguindo os princípios de uma arquitetura REST.

Tal como foi explicado anteriormente, este módulo é necessário devido à plataforma em que se encontram registados os utilizadores que usufruirão do serviço. Isto faz com que este módulo tenha de incluir alguma funcionalidade suplementar ao invés de funcionar apenas como *"proxy"*. A principal funcionalidade suplementar do mesmo consiste em complementar os objetos devolvidos pelo módulo NotesAthena, com informações do utilizador, tais como o nome de utilizador ou o caminho para a fotografia no servidor.

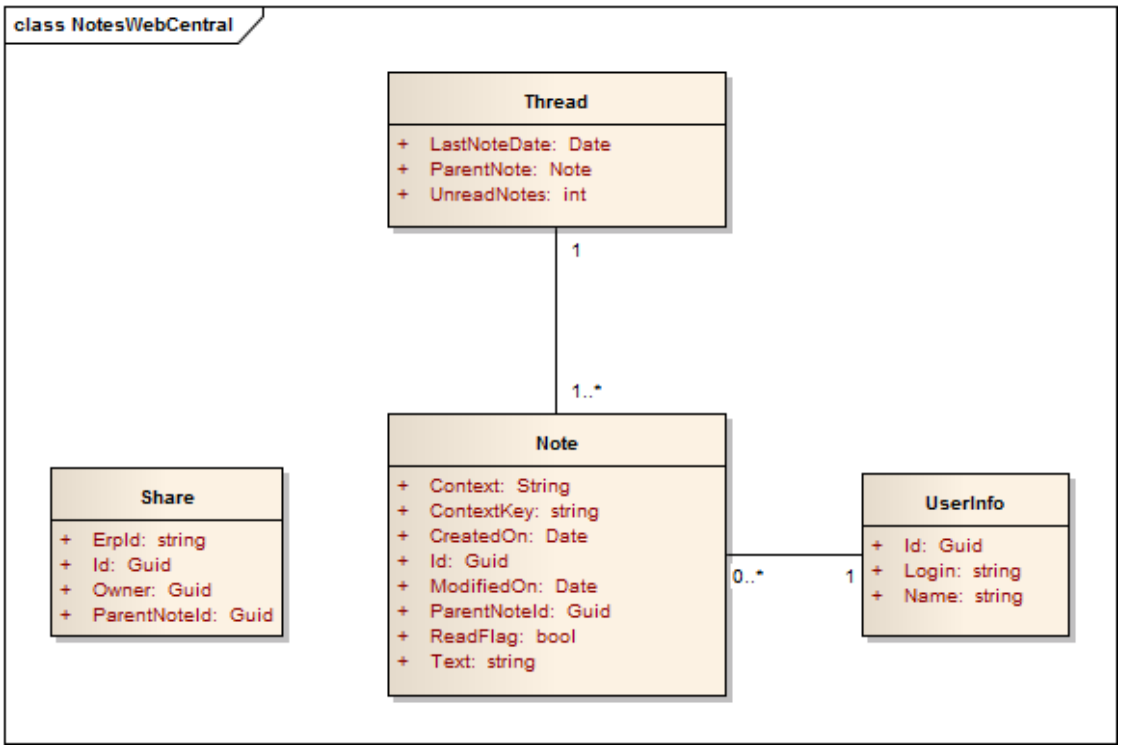
Neste módulo foram ainda implementados as classes para os objetos de transferência de dados, o que permitiu reduzir a complexidade necessária do lado dos clientes. Tendo em conta que a apresentação de conteúdos em ambos segue lógicas muito semelhantes, a implementação de funcionalidade numa camada inferior é vantajosa, pois evita a criação de código semelhante em sítios diferentes para servir propósitos iguais.

5.2.1 Objetos de Transferência de Dados NotesWebCentral

Foram então desenhadas classes para criação de objetos de transferência de dados *(DTO), as quais estão* representadas na **Figura 4.6**.

A classe **Thread** é uma classe que encapsula uma lista de Notas relacionadas, tornando esta lista num tópico de conversa. Esta permitiu ainda incluir algumas propriedades úteis para a camada de apresentação, como o número de notas não lidas de um conjunto de notas relacionadas, a data da última nota adicionada ou ainda a Nota pai (ParentNote), que inicia um tópico.

Figura 5.3 – Diagrama de Classes dos DTO *(DataTransferObjects)*



A construção destes objetos neste módulo reduz de forma significativa a complexidade necessária do lado das aplicações cliente, sendo que os objetos são devolvidos pelo serviço, já preenchidos com os dados necessários para as camadas de apresentação.

A classe **Note** representa a classe já modelada no módulo NotesAthena, no entanto, inclui também dados adicionais. A principal propriedade adicional é acerca da informação do utilizador com a associação à classe **UserInfo**. Esta classe contém as informações acerca do login (tipicamente o endereço de correio eletrónico) e o nome do utilizador. Além desta propriedade, inclui ainda a propriedade ReadFlag, que indica se a nota já foi lida ou não pelo utilizador.

Relativamente à classe **Share**, esta serve para a criação de objetos com a informação das partilhas de uma Thread, incluindo o Id da Nota pai partilhada e os dados do utilizador (UserInfo) a que a partilha pertence.

5.2.2 Serviços NotesWebCentral

A camada de serviços do módulo NotesWebCentral é a camada de serviços expostos para acesso exterior, ou seja, é nesta que as aplicações cliente consomem os serviços desenvolvidos.

A arquitetura de serviços implementada nesta camada seguiu, tanto quanto possível, os princípios de uma arquitetura REST, como a utilização do protocolo http como método de transporte, fazendo uso dos métodos *GET, PUT, POST, DELETE* e tentando orientar os serviços a nomes (recursos). A API desenhada foi a seguinte:

Listagem 5.2 – API de serviços NotesWebCentral

```
/// Threads
GET           /threads/search?mck={masterContextKey}
POST          /threads
DELETE        /threads/{key}

/// Notes
GET           /notes/search?q={query}
POST          /notes
DELETE        /notes/{key}
PUT           /notes/{key}

/// Shares
GET           /shares
POST          /shares
```

Cada um destes métodos pode usar um ou mais serviços do módulo NotesAthena. A título de exemplo, o pedido GET enviado a `/threads` pode incluir ou não uma masterContextKey. Caso tenha essa informação no pedido, é enviada a lista com as *threads* associadas ao contexto da chave passada, ou seja, usando o serviço Athena *GetNotesByMasterContextKey*, caso contrário, é usado o serviço *GetNotesByUserId.* Não é necessário fornecer o ID do utilizador, uma vez que essa informação está guardada no serviço após a autenticação das aplicações cliente.

Outro exemplo de utilização desta API é por exemplo a criação de uma nova nota. Para isso é necessário fazer um pedido POST ao endereço `/notes`.

Listagem 5.3 – Conteúdo de um POST ao serviço /notes

```
POST https://pribss.com/services/notes HTTP/1.1
Accept: application/json, text/javascript,
Content-Type: application/json; charset=utf-8
X-Requested-With: XMLHttpRequest
Referer: https://www.clouddevelopment.com/userspace/Notes.aspx
Accept-Language: pt-PT,pt;q=0.5
Accept-Encoding: gzip, deflate
User-Agent: Mozilla/5.0
Host: www.clouddevelopment.com
Content-Length: 91
DNT: 1
Connection: Keep-Alive
Cache-Control: no-cache
Cookie: ASP.NET_SessionId=rmytigmk1uxtup23dl0jobyn

{"ParentNoteId":"9dec66ed-a76b-46d3-8644-bcd22583907b","Text":"Esta é uma
nota de teste."}
```

Ao receber este pedido, o módulo NotesWebCentral invoca o método *PostNote* (**Listagem 5.4**), que por sua vez trata da comunicação com o módulo NotesAthena.

Este excerto de código (**Listagem 5.4**) é o código implementado no NoteManager do módulo NotesWebCentral. Este método é chamado, numa primeira fase, por uma classe do componente Services que adiciona a informação EngineContext, necessária para efetuar a comunicação com o módulo NotesAthena. Adiciona também o Guid AthenaOrganizationId que irá permitir identificar o utilizador que efetuou o POST da nota. Os dados recebidos em JSON foram já convertidos no DTO Note, sendo estes usados para criar o DataContract a enviar ao serviço Athena. Caso a resposta tenha sucesso, é devolvido um objeto semelhante, mas com mais propriedades preenchidas, como a data de criação no servidor. Isto permite aos clientes saberem que a inserção ocorreu com sucesso.

Listagem 5.4 – Excerto do método C# PostNote da classe NoteManager

```
/// <summary>
/// Posts the note.
/// </summary>
/// <param name="context">The Engine context.</param>
/// <param name="athenaOrganizationId">The athena organization id.</param>
/// <param name="oneNote">The note to Post.</param>
/// <returns>The Note posted</returns>
/// <exception cref="System.Exception">Problem posting Note</exception>
public Note PostNote(EngineContext context, Guid athenaOrganizationId, Note oneNote)
{
    try
    {
        using (INoteServiceChannel proxy = ChannelBuilder.BuildNoteChannel(context))
        {
            NoteDataContract theNote = oneNote.ToExternalDataContract();

            theNote.Owner = context.UserID;

            PostNoteRequest postNoteRequest = new PostNoteRequest()
            {
                OrganizationId = athenaOrganizationId,
                TenantId = context.OrganizationID,
                note = theNote,
                context = oneNote.Context,
                contextKey = oneNote.ContextKey
            };

            PostNoteResponse response = proxy.PostNote(postNoteRequest);

            if (response != null && response.Result != null)
            {
                // Translate to the Note Internal DataContract
                var noteResponse = response.Result.ToInternalDataContract();

                // Fill the NoteDataContract with the info of ERPContext
                noteResponse = NotesUtilities.FillContextInfo(context, athenaOrganizationId,
noteResponse);

                // Fill the OwnerInfo of the NoteDataContract
                noteResponse.OwnerInfo = NotesUtilities.GetUserInfo(context, context.UserID);

                return noteResponse;
(…)
```

5.3 Aplicações Cliente

Para este projeto foram implementados dois clientes que consomem os serviços desenvolvidos. Um deles, uma *Windows Store App,* o outro, o controlo WebCentral a ser usado no ERP Primavera e no UserSpace Primavera.

Nesta secção serão apresentados os principais detalhes da implementação feita.

5.3.1 Windows Store App

Na secção 2.4 já foi feita uma pequena contextualização das aplicações desenvolvidas para a loja virtual de aplicações da Microsoft, cujo objetivo é "fornecer" o seu ecossistema de

sistemas operativos. Nesta secção são explicadas as decisões de implementação tomadas relativamente a esta aplicação cliente.

No que concerne à implementação deste tipo de aplicações, a primeira decisão tomada foi acerca das linguagens de programação a usar.

Linguagem de Programação

A Microsoft oferece várias alternativas de implementação (**Listagem 5.5**), sendo que cada uma tem as suas vantagens e desvantagens.

Listagem 5.5 – Alternativas de implementação das Windows Store Apps

Linguagem de Programação	Tecnologia de Apresentação
C#/ Visual Basic	XAML
JavaScript	HTML5
C++/CX	Interoperabilidade de XAML, DirectX e XAML/DirectX

Adaptado de Microsoft (2013)

Oficialmente, a Microsoft não aconselha nenhuma das alternativas em detrimento de outra, sendo que apenas garante suporte igual para todas elas e aconselha que a escolha seja baseada na experiência do programador em alguma das alternativas e, ainda, na conveniência de cada uma para o tipo de aplicação a desenvolver. Normalmente, a utilização de C++/DirectX é orientada ao desenvolvimento de aplicações que necessitem de um maior desempenho gráfico, nomeadamente jogos, o que não é o caso. No caso da escolha entre C# e VB, tendo em conta que o método de programação seria semelhante, a escolha seria C#, devido à experiência já obtida na linguagem. A escolha final resumia-se então a duas alternativas: C# e Javascript.

Por um lado, a linguagem Javascript é uma das linguagens atualmente mais usadas para programação de aplicações web, o que permitiria posteriormente alavancar o conhecimento obtido da mesma noutros projetos, mas, por outro lado, é uma linguagem pouco robusta, com tipagem fraca e onde é mais fácil cometer erros. O Javascript é uma linguagem interpretada, o que faz com que muitos erros apareçam apenas em *run-time.*

O C#, por outro lado, é uma linguagem considerada mais robusta, mais fácil de testar e, além disso, é uma linguagem compilada, o que geralmente significa melhor desempenho. Outra das razões encontradas em favor do C# prende-se com a API para lidar com chamadas a métodos assíncronos, que em C# envolvem menos código e onde é menos provável o

programador cometer erros. Os métodos assíncronos são um dos requisitos fundamentais nesta aplicação devido às chamadas a serviço *web*. A *framework* .NET, para Windows Store Apps, requer que todas as chamadas a serviços sejam assíncronas, para não obrigar a aplicação a bloquear enquanto espera pela resposta do serviço.

Pesando os prós e contras de cada uma das alternativas, a escolha recaiu sobre o C#, que recorre a XAML para a construção da camada de apresentação.

Implementação

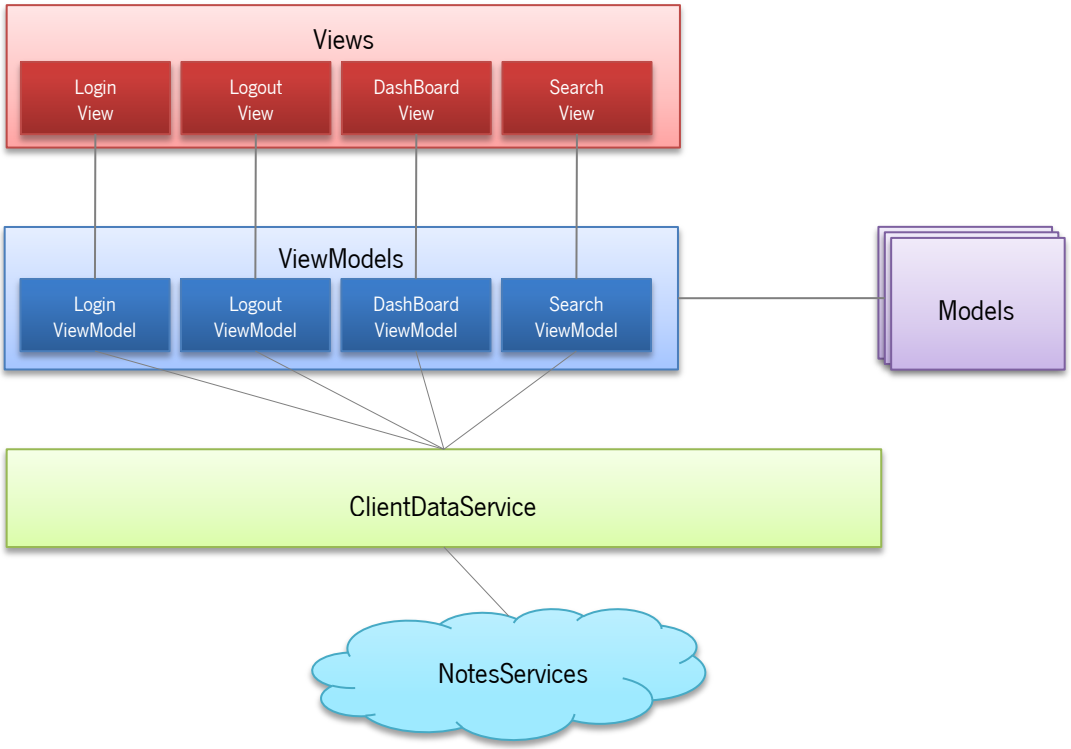
Nesta secção será detalhada a implementação da *Windows Store App*, explicando a estrutura construída e a forma como funciona.

A *Windows Store App* é composta basicamente por quatro *Views* diferentes e respetivos *ViewModels* (Figura 5.4):

- **Login:** é a vista onde o utilizador efetua a sua autenticação. Sem a autenticação não é possível utilizar a aplicação.
- **Logout:** vista usada no *flyout* (espécie de janela que aparece lateralmente na aplicação) das definições da aplicação, responsável por permitir ao utilizador terminar sessão.
- **Dashboard:** ponto central da aplicação. É onde o utilizador pode consultar as notas, responder a uma das notas, ou ainda remover uma das notas que tenha escrito. Pode também consultar uma grelha com os dados do contexto a que as notas pertencem.
- **Search:** vista que apresenta os resultados da pesquisa de notas da aplicação. Os termos de pesquisa são introduzidos usando o mecanismo de pesquisa do Windows, que envia o pedido de pesquisa ao *ViewModel, o* qual obtém os resultados apresentados pela *View.*

Os *Models* usados na aplicação são cópias dos objetos de transferência de dados do módulo NotesWebCentral.

Figura 5.4 – Diagrama da arquitetura da aplicação Windows



Os *ViewModels* efetuam os pedidos que necessitam à classe *ClientDataService,* que é responsável por comunicar com os NotesServices (NotesWebCentral). Esta classe trata da lógica necessária para a chamada aos serviços e, além disso, recorre a um método de de-serialização dos objetos JSON devolvidos, construindo objetos utilizando os *Models* criados a partir dos DTO. No caso dos pedidos POST, os objetos seguem o percurso contrário, sendo serializados em JSON e enviados ao serviço.

Como exemplo, será aqui demonstrado como é feita a ligação entre aquilo que é apresentado numa *View* e o serviço.

Na secção 0 Padrão MVVM (Model – View – ViewModel), foi já referido que é através do mecanismo de *data-binding* que os dados são apresentados no ecrã. Isto é feito explicitamente no código XAML, como se pode ver na **Listagem 5.6**, onde se encontra um exemplo em que o texto apresentado num controlo *TextBlock* está a efetuar *binding* à propriedade "Total" do *ViewModel.*

Listagem 5.6 – Exemplo de data-binding

```
<TextBlock Text="{Binding Total}"/>
```

Na **Listagem 5.7**, está representada a caixa de texto onde é escrito o conteúdo de uma nota a enviar como resposta. De seguida encontram-se dois botões, um de envio e outro para cancelar a operação. Tipicamente isto é feito recorrendo a um mecanismo de *Commands* que efetua o *binding* da propriedade a um método do *ViewModel.* Neste trabalho, no entanto, foi usada uma biblioteca chamada *Caliburn.Micro,* que efetua esse mapeamento automaticamente a partir do nome do elemento por convenção. Neste caso, ao carregar no botão *PostNote,* é executado o método correspondente no *ViewModel.*

Listagem 5.7 – Excerto de código da *DashboardView.xaml*

```
<!-- New Note Text Box -->
<TextBox x:Name="NewNoteTextBox"
         AcceptsReturn="True"/>

<Grid Grid.Row="1"
      Margin="0,5,0,0">
      <Grid.ColumnDefinitions>
        <ColumnDefinition Width="*"/>
        <ColumnDefinition Width="Auto"/>
      </Grid.ColumnDefinitions>

      <Button x:Name="PostNote"
              x:Uid="SaveNewNote"
              Content="&#xE10B;”/>

      <Button x:Name="CancelNewNote"
              x:Uid="CancelNewNote"
              Content="&#xE10A;"/>
</Grid>
```

De seguida será demonstrado esse mesmo método no *ViewModel* correspondente a esta *View.*

Listagem 5.8 – Método PostNote do DashboardViewModel

```
/// <summary>
/// Posts a new note on the current Thread
/// </summary>
private async void PostNote()
{
    // Check if there is text in the textbox
    if (this.NewNoteTextBox != null && this.NewNoteTextBox.Trim() != string.Empty)
    {
        // Change the animation property state
        this.IsSaving = true;

        // Hide the textbox
        this.IsNewNoteTextBoxVisible = false;

        // Create the note to POST
        Note n = new Note
        {
            Text = this.NewNoteTextBox,
            ParentNoteId = this.SelectedThread.ParentNote.Id,
            ContextId = this.relatedNotes.Last().ContextId
        };

        // POST to the service
        Note result = await this.dataService.PostNote(n);

        // If the result was successful
        if (result != null)
        {
            // Add the note to the current related Notes List
            this.RelatedNotes.Add(result);

            // Also add to the Thread List (in memory list of all notes)
            this.SelectedThread.ThreadNotes.Add(result);

            // Change the selected related note to this last one inserted
            this.SelectedRelatedNote = this.RelatedNotes.Last();

            // Clear the new note text box
            this.NewNoteTextBox = string.Empty;
        }
        else
        {
            // Get the resource loader
            var loader = new Windows.ApplicationModel.Resources.ResourceLoader();

            // Create the message dialog and set its content
            var messageDialog = new MessageDialog(loader.GetString(Helper.NOTEPOSTERROR));

            // Show the message dialog
            await messageDialog.ShowAsync();
        }

        this.IsSaving = false;
    }
}
```

No excerto de código acima (**Listagem 5.8**), é possível verificar um exemplo do *ViewModel* que efetua o POST de uma nova nota. Para isso recorre à classe *ClientDataService* (no código representada pela variável dataService), executando o método *PostNote(Note n).*

A classe *ClientDataService* (**Listagem 5.9**), por sua vez, tratará de efetuar os passos necessários para serializar o objeto em JSON e respetivos passos necessários para a correta comunicação com o serviço. A comunicação com este é feita através do método privado *PostRestAsync(),* que trata dos pormenores de comunicação comuns a todos os pedidos feitos ao serviço REST do módulo NotesWebCentral.

Listagem 5.9 – Método PostNote da classe *ClientDataService*

```
/// <summary>
/// Posts a note to the Service
/// </summary>
/// <param name="note">The Note to post</param>
/// <returns>
/// If successful, returns the note posted
/// </returns>
public async Task<Note> PostNote(Note note)
{
    Log.Info("Posting Note");

    // Serialize the note as JSON
    StringContent content =
        new StringContent(
                Helper.JsonSerializer<Note>(note), Encoding.UTF8, "application/json");

    content.Headers.ContentType = new MediaTypeHeaderValue("application/json");

    HttpResponseMessage httpResponse = await this.PostRestAsync("Notes", content);

    string responseContent = await httpResponse.Content.ReadAsStringAsync();

    Note responseNote = null;

    // If the response is not null
    if (!string.IsNullOrEmpty(responseContent))
    {
        // Deserialize the response to the corresponding Share
        responseNote = Helper.Deserialise<Note>(responseContent);
    }

    Log.Info("COMPLETED: Posting Note");

    return responseNote;
}
```

5.3.2 Controlo Web NotesWebCentral

Um dos requisitos do projeto era a criação de um controlo de apresentação *web* WebCentral que pudesse ser apresentado em dois pontos diferentes: no portal *UserSpace* e diretamente no ERP. Nesta secção são apresentadas as principais decisões tomadas na implementação deste componente.

Implementação

Este componente *web* é um controlo que se encontra integrado no *UserSpace.* Apesar desta integração, o mesmo componente é reutilizado no ERP de forma transparente, sem ser necessário fazer qualquer duplicação de código ou processo de *deploy* noutro ambiente. A razão

para tal transparência é que o ERP apresenta o mesmo portal, usando um controlo do ERP que acede a um URL, tal como um navegador *web* acede.

Uma vez que o controlo se encontra alojado no *UserSpace,* havia poucas decisões a tomar relativamente à implementação, dado que teria de integrar com este produto. A implementação foi então feita usando a linguagem JavaScript e recorrendo a HTML para estruturar a camada apresentação. Os serviços expostos pelo módulo foram consumidos em funções JavaScript, preenchendo o conteúdo do componente com os resultados devolvidos.

Listagem 5.10 – Função PostNote do controlo NotesWebCentral

```
// Posts a new Note
function postNote() {

    // Get the Text to Post
    var textToPost = $("#<%=this.ClientID%>_divNote").find('textarea[name="txtN"]')[0].value;

    // Create the new note Object to Post
    var newNoteObject = {
        ParentNoteId: NotesVars.CurrentParentNoteId,
        Context: NotesVars.Context
        Text: textToPost
    };

    if (newNoteObject.Text.length > 2) {

        // Address where to Post the Note
        var address = '<%= this.ServiceAddress %>/Notes';

        // Block the UI while the Post is being done
        blockUIelem($('.divRelatedNotesContainer'));

        // Do the AJAX Post
        $.ajax({
            type: "POST",
            url: address,
            data: JSON.stringify(newNoteObject),
            contentType: "application/json; charset=utf-8",
            dataType: "json",

            // On sucess, do the following:
            success: function () {

                // Clear the textBox
                $('[name="txtNote"]').val('');

                // Refresh the list of the RelatedNotes
                getRelatedNotes(NotesVars.CurrentParentNoteId);

                // Get the RelatedNotes div
                var objDiv = document.getElementById("<%=this.ClientID%>_NotesContainer");

                // Scroll to the bottom of the RelatedNotes div
                objDiv.scrollTop = objDiv.scrollHeight;

                // UnBlock when the UI POST has been completed
                unblockUIelem($('.divRelatedNotesContainer'));
            }
        });
    }
}
```

Um dos aspetos que facilitou o trabalho foi o facto de os serviços devolverem objetos JSON (JavaScript Object Notation), que podem ser usados de forma nativa pelo JavaScript, pelo que não houve necessidade de converter os objetos. No excerto de código acima (**Listagem 5.10**) pode ver-se o exemplo do método JavaScript desenvolvido para efetuar o POST de uma nota. As diferenças no algoritmo, comparativamente com a versão da aplicação Windows, são notórias. Neste caso, o controlo encontra-se alojado no UserSpace, pelo que, questões de autenticação estavam já tratadas pelo próprio portal. Outra diferença que se pode detetar é a falta de uma organização MVVM, uma vez que não fazia parte do âmbito deste trabalho, o estudo deste tipo de arquitetura em controlos *web* JavaScript. O método captura os dados da interface e invoca diretamente o serviço de *posting* da nota com os dados capturados. Recebendo a resposta, faz a respetiva atualização da lista de notas.

Seria interessante no futuro tentar aplicar neste controlo uma arquitetura MVVM semelhante à aplicação Windows.

Contexto

Os maiores desafios que se colocaram na construção deste componente foram: encontrar uma forma de comunicação entre este e o ERP, e definir uma forma padrão de guardar o contexto a que as notas se encontram associadas. Estes desafios eram dos pontos mais importantes, uma vez que sem o contexto de negócio associado às notas, esta tornar-se-ia *mais uma* aplicação de notas, como muitas outras. A comunicação era então essencial, dado que era fundamental ocorrerem duas coisas. Em primeiro lugar, o componente apenas deveria mostrar as notas associadas ao contexto ativo no ERP, ou seja, ao documento sob o qual se está a trabalhar no ERP. Além disso, era necessário passar para o lado do controlo, uma serialização do documento que permitisse posteriormente recriar uma representação sua na *Windows Store App* e no componente web.

O problema da comunicação ficou resolvido com a personalização de um *plugin* para o ERP, que permite a chamada de funções Javascript da página carregada pelo controlo do ERP, que apresenta páginas Web. Assim, quando um utilizador se encontra a trabalhar, por exemplo, numa fatura, e acede à funcionalidade de notas, é invocada uma função Javascript que apenas carrega as notas relacionadas com aquela fatura. A invocação desta função permite também passar ao componente a serialização construída do documento. Esta fica pronta em memória

para ser usada quando o utilizador cria uma nota e faz POST da mesma para o serviço de criação de notas.

Listagem 5.11 – Exemplo da estrutura XML que guarda um contexto

```
<?xml version="1.0" encoding="UTF-8"?>
<entity>
  <master>
    <fields>
      <field>
        <property>Data</property>
        <title>Data</title>
        <value>19-07-2013</value>
      </field>
      ...
  </master>
  <detail>
    <model>
      <id>CONST_ARTIGO</id>
      <fields>
        <field>
          <title>Artigo</title>
          <description>Artigo</description>
          <length />
          <editable />
          <type>string</type>
        </field>
        ...
      </fields>
    </model>
    <lines>
      <line>
        <field>
          <property>Artigo</property>
          <title>Artigo</title>
          <value>A0001</value>
        </field>
        ...
      </line>
    </lines>
  </detail>
</entity>
```

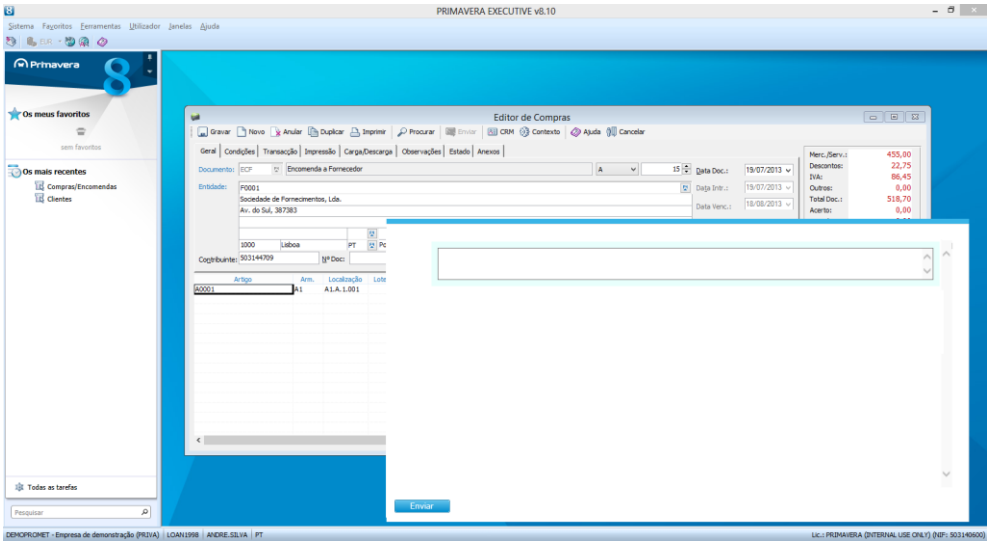
Relativamente à serialização dos documentos, foi desenvolvida uma estrutura XML padrão (**Listagem 5.11**) que pudesse ser usada pelas aplicações para representar os documentos sobre os quais as notas são escritas. No exemplo acima, foram retirados alguns elementos repetidos para não tornar o exemplo demasiado longo. Esta é uma estrutura que segue uma lógica *MasterDetail – Detail*, e é construída numa função no ERP que serializa o objeto em causa. Neste projeto, esta construção, como prova de conceito, apenas é feita num tipo de documentos, no entanto, espera-se que a mesma estrutura XML sirva para vários tipos de documentos, sendo apenas necessário implementar a funcionalidade que serializa o objeto para cada um dos tipos de documentos.

5.3.3 Interação com o utilizador

Nesta secção serão apresentadas as capturas de ecrã para os casos de uso especificados.

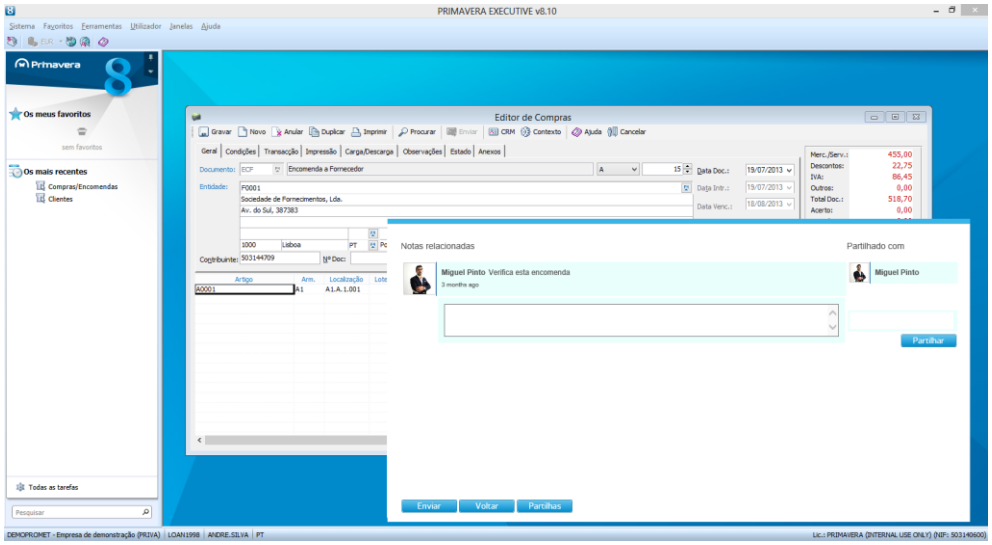
Tipicamente, a interação começa no ERP, com o utilizador a criar um documento. Ao aceder ao menu de contexto de notas aparece o formulário para criação de um novo Tópico associado a este documento (**Figura 5.5**).

Figura 5.5 – Captura de ecrã "Criar Tópico"



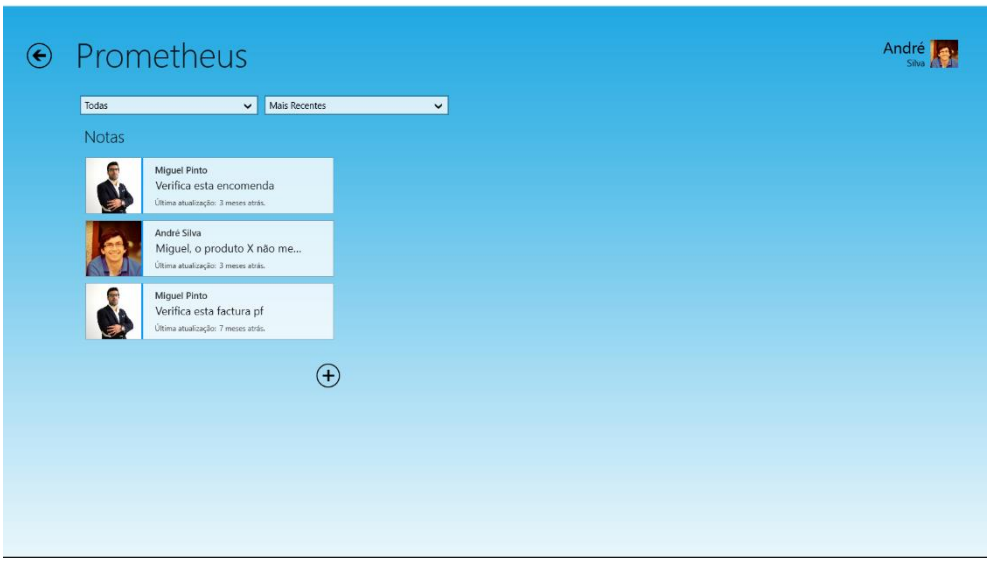
A criação de um Tópico implica a criação de uma nota que será a nota pai. A esta nota segue também associado o contexto do documento ativo. De seguida, o utilizador que cria o Tópico pode partilhá-lo com outros utilizadores.

Figura 5.6 – Captura de ecrã "Partilhar Tópico"



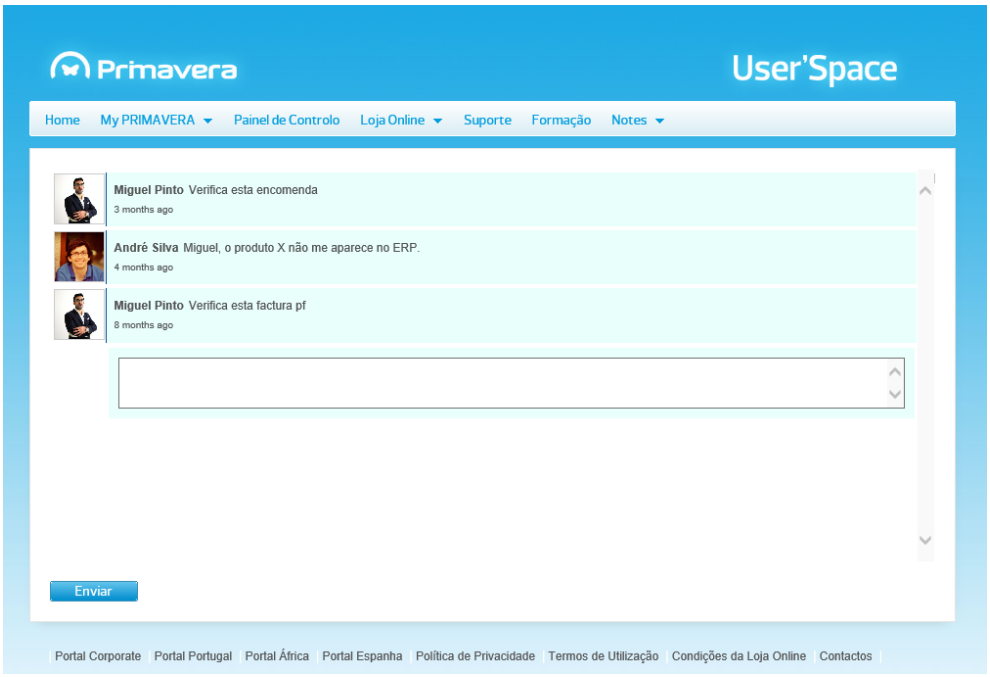
A partir deste momento, os utilizadores com os quais foi partilhado o Tópico, têm acesso, em qualquer aplicação cliente, às notas relacionadas a este Tópico. Por exemplo, na **Figura 5.7**, outro utilizador, além do criador do Tópico, pode ver a lista de Tópicos a que tem acesso, incluindo aquela que acabou de ser partilhada consigo.

Figura 5.7 – Captura de ecrã "Consultar Tópicos Partilhados"



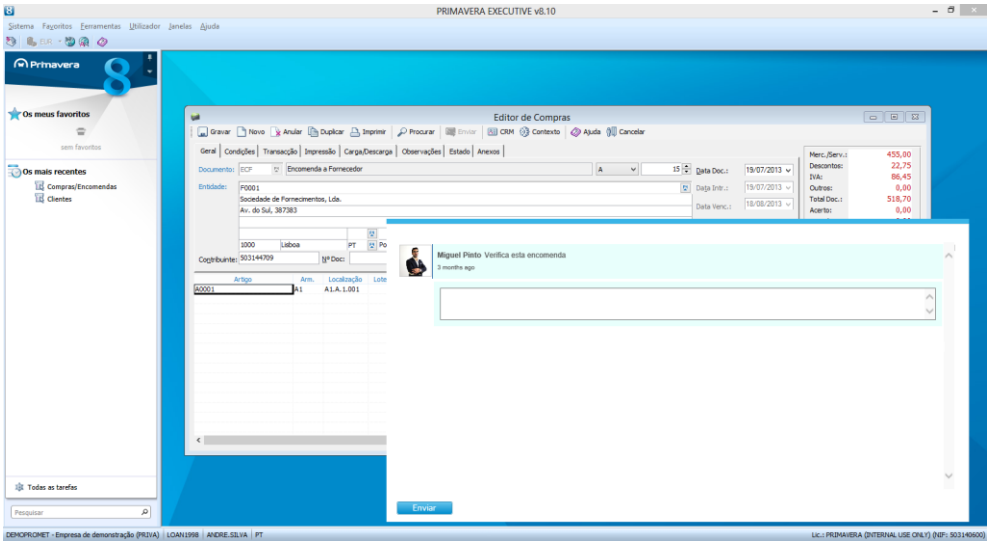
Uma lista com a nota pai de cada Tópico é apresentada à esquerda. No caso do UserSpace, a lista apresentada é semelhante. O controlo desenvolvido e implementado no UserSpace apresenta os mesmos Tópicos que na aplicação Windows (**Figura 5.8**).

Figura 5.8 – Captura de ecrã "Consultar Tópicos Partilhados - UserSpace"



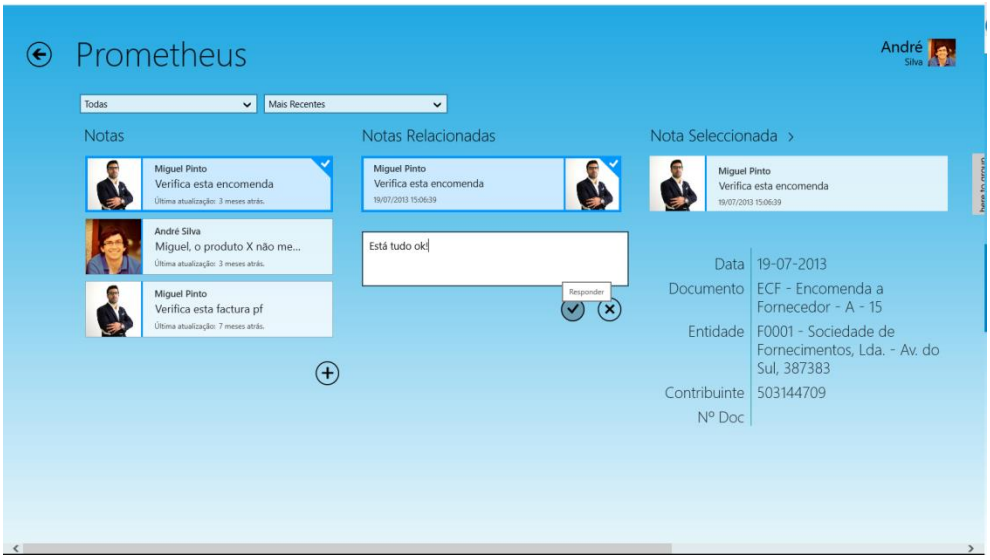
No cenário em que o utilizador está no ambiente do ERP, a lista de Tópicos apresentada, é filtrada pela MasterContextKey, apresentando apenas os Tópicos associados ao documento ativo (**Figura 5.9**).

Figura 5.9 – Captura de ecrã "Consultar Tópicos Partilhados – ERP"



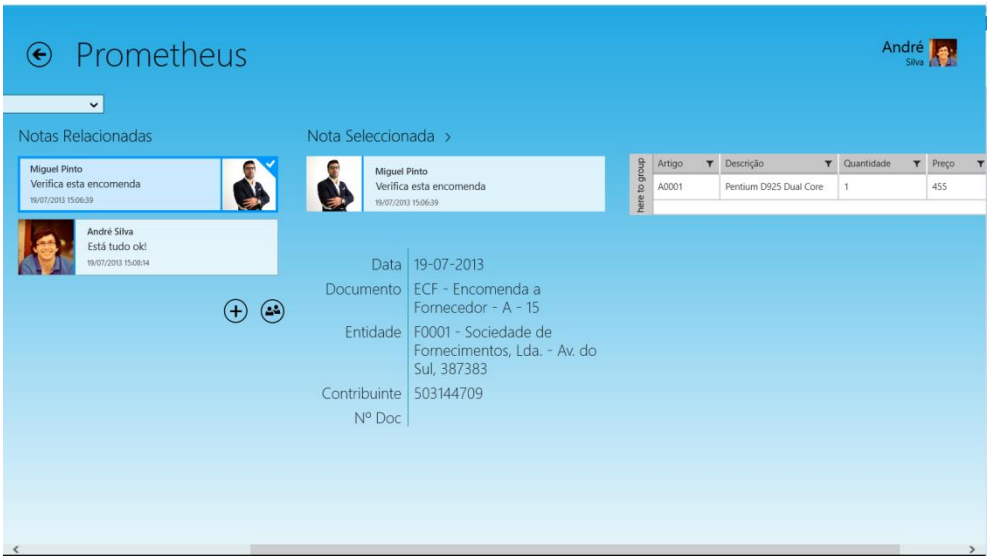
Cada Tópico tem uma lista de notas relacionadas. O outro utilizador pode então responder no Tópico partilhado com ele (**Figura 5.10**).

Figura 5.10 – Captura de ecrã "Responder a Tópico"



Além da resposta pode também ver o contexto da nota, ou seja, pode ver a representação reduzida do documento origem à qual foi associada esta nota.

Figura 5.11 – Captura de ecrã "Consultar Notas Relacionadas"



Com a resposta do segundo utilizador, está então concluído o cenário de partilha de notas a partir do ERP e a respetiva utilização da aplicação da *Windows Store* como aplicação que consome o serviço.

Além da interação demonstrada acima, foram ainda implementadas as funcionalidades de pesquisa de notas e de filtragem e ordenação de notas (**Figura 5.12**, **Figura 5.13**).

Figura 5.12 – Captura de ecrã "Pesquisar Notas"

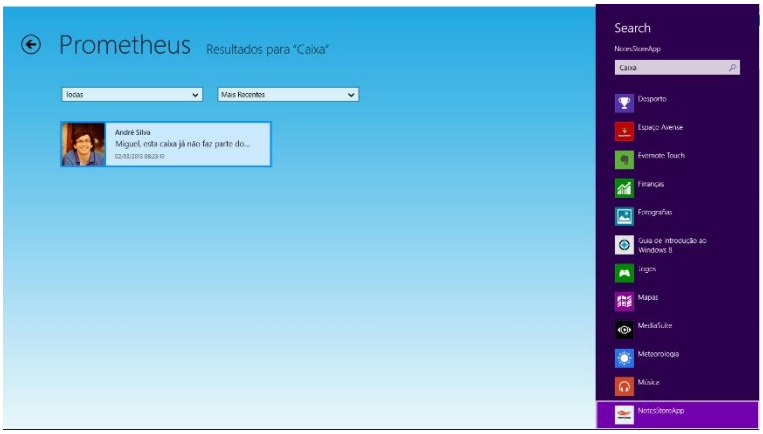
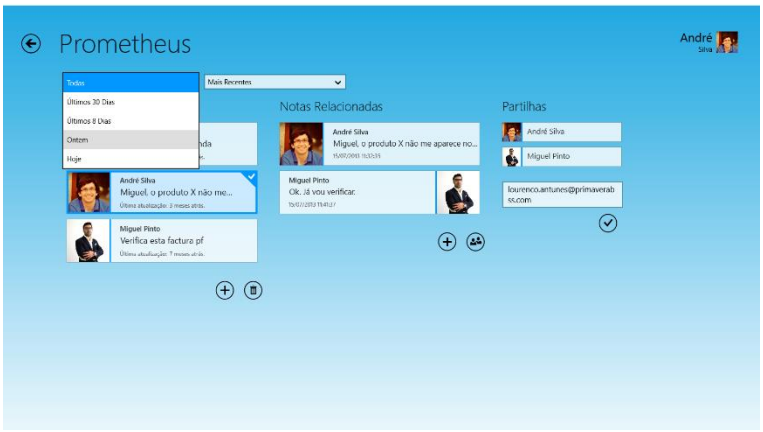


Figura 5.13 – Captura de ecrã "Filtrar Notas"



Nesta secção foi demonstrada a interação do utilizador com as aplicações desenvolvidas. Foi seguido o fluxo de uma interação típica com o ERP e respetivas aplicações. Todos os casos de uso, com exceção do "Pesquisar Notas" e "Filtrar e Ordenar Notas", foram implementados tanto na aplicação *Windows Store,* como no controlo web do módulo NotesWebCentral.

Capítulo 6

Conclusões

O desenvolvimento de aplicações usando uma arquitetura orientada a serviços tem sido uma das metodologias mais usadas nos últimos anos no desenvolvimento de *software*. Anteriormente, durante o *boom* dos computadores pessoais, a construção de aplicações de software, seguiu uma arquitetura tipicamente local e isolada de outras aplicações e dispositivos. No entanto, esta abordagem foi perdendo o fulgor de outros tempos. Hoje em dia, o mundo está em permanente ligação através da internet, e, a partir daí, surgiu uma multiplicidade de dispositivos e formas de consumir a informação para além do típico computador pessoal. Esta situação verifica-se também no caso das aplicações de âmbito empresarial, em que a informação é igualmente espalhada por diversos dispositivos. A comunicação entre aplicações e a troca de dados através da rede é quase uma obrigatoriedade no panorama atual do desenvolvimento de *software*.

Esta comunicação com o mundo faz com que seja necessária uma abordagem diferente na construção de aplicações, como o uso de uma arquitetura orientada a serviços. A utilização de serviços *web* obriga a uma mudança de paradigma no que diz respeito à forma como são criadas as aplicações. Enquanto no passado as aplicações tradicionais eram executadas num ambiente local e muito controlado, com a utilização de serviços que atravessam a rede, perdeu-se algum desse controlo. A transmissão de dados via rede obriga a que as aplicações que fazem uso de serviços sejam planeadas de raiz, de forma a ter em conta situações que anteriormente não tinham que antever, como, por exemplo, a possibilidade de uma comunicação ser interrompida, perdendo-se parte dos dados ou, ainda, o facto de uma resposta a uma ação poder demorar mais tempo que o esperado.

O tema desta dissertação surgiu da necessidade de investigar este tipo de abordagem ao desenvolvimento de *software*, no âmbito de um projeto desenvolvido na empresa Primavera BSS.

Para desenvolver esta dissertação foi necessário fazer um estudo acerca da arquitetura orientada a serviços, nomeadamente de que forma é possível implementar a mesma no âmbito de serviços acessíveis a partir da rede (serviços *web*). A natureza deste trabalho permitiu ainda aplicar duas metodologias diferentes na construção de serviços. Por um lado, a construção de serviços seguindo uma abordagem RPC *(Remote Procedure Call)* com recurso à *framework* Microsoft WCF *(Windows Communication Foundation)* e respetivas mensagens SOAP. Por outro lado, a construção de serviços usando uma abordagem REST. Esta dualidade permitiu retirar algumas ilações.

A utilização da abordagem WCF SOAP, no âmbito dos serviços implementados no módulo NotesAthena, demonstrou algumas características que podem ser vantajosas em alguns cenários. Uma das vantagens foi encontrada na construção do módulo NotesWebCentral, que consumiu diretamente estes serviços na camada de servidor. Uma vez que também este módulo foi construído com tecnologia Microsoft .NET, é fácil adicionar ao módulo as referências necessárias ao serviço e, logo de seguida, os métodos expostos pelo serviço NotesAthena estão disponíveis no código, como se apenas tivesse sido adicionada uma nova classe ou biblioteca. Isto acontece graças à WSDL *(Web Services Description Language),* que define de forma clara o contrato da funcionalidade oferecida pelo serviço. No entanto, para que tal aconteça, é necessário que a *framework* WCF esconda um mecanismo de configuração automática relativamente complexo. Isto torna o código opaco e dependente da ferramenta, o que faz com que a correção de problemas possa ser custosa. Além disso, a interoperabilidade deste tipo de serviços com tecnologia que não seja Microsoft .NET é baixa, o que derrota uma das principais vantagens da utilização de serviços *web.* Apesar de neste trabalho não ter sido possível testar com tecnologia de outro fabricante, o mecanismo de comunicação e respetiva configuração é complexo o que pode torná-lo difícil de implementar manualmente. De facto, nem mesmo as aplicações da loja de aplicações Windows, suportam todos os protocolos de comunicação disponíveis na *framework* WCF, o que revela que este protocolo deve apenas ser usado em cenários específicos. A abordagem WCF SOAP tem ainda a vantagem de poder ser usada sobre diferentes protocolos de comunicação além do http, como o SMTP ou o TCP, no entanto, esta característica não foi objeto de análise nesta dissertação.

Relativamente à utilização da abordagem REST, conclui-se que é uma arquitetura que facilita a interoperabilidade com diferentes tecnologias. De facto, a utilização do protocolo http encontra-se massificada, pelo que se torna simples criar uma aplicação em qualquer tecnologia

que consuma um serviço REST. Além disso, não exige um sistema complexo de comunicação que recorra a WSDL, sendo a comunicação feita usando http e com recurso à sua interface, a qual já é conhecida. Com efeito, para a utilização de um serviço REST é apenas necessário um mecanismo de serialização de objetos, num formato padrão como JSON ou XML. No entanto, a não utilização de WSDL obriga que seja conhecida a estrutura dos objetos de transporte de dados bem como a sua correta definição manual, no lado das aplicações cliente. A abordagem REST obriga ainda à definição de uma API de recursos acessíveis via URI, cuja definição influencia a sustentabilidade do serviço.

Era também objetivo desta dissertação estudar o cenário de partilha de anotações sobre documentos, para facilitar a comunicação entre pessoas que usem o ERP nos processos de negócio. Foi feito um levantamento de requisitos e analisados alguns cenários de utilização. Posteriormente foram implementados os requisitos levantados, não sendo possível, nesta dissertação, tirar ilações acerca do objetivo da facilitação da comunicação.

Um dos objetivos traçados foi ainda o estudo do desenvolvimento de aplicações da loja Windows e respetiva conceção de uma aplicação. Este tipo de aplicações segue uma tendência no âmbito das aplicações de uso doméstico/pessoal, no entanto, é muito diferente das tradicionais, no que diz respeito ao uso com rato e teclado. Apesar de não ser obrigatório, são orientadas para uso em dispositivos táteis como *tablets*, pelo que devem ser construídas com esse tipo de utilização em mente.

Concluiu-se ainda que estas não têm como objetivo substituir as suas predecessoras, mas sim complementá-las. Não é possível construir aplicações deste tipo com a complexidade de um ERP, mas é possível construir aplicações que complementem processos de negócio ou que incluam uma parte destes.

Este tipo de aplicações está intimamente ligado a um ambiente tipicamente orientado a serviços, pelo que, a qualidade de uma aplicação depende, não só da sua qualidade enquanto aplicação, mas também, em grande parte, dos eventuais serviços que a alimentam. Estas podem ser aplicações dedicadas a um serviço, mas também podem de igual forma orquestrar diversos serviços, como o caso das aplicações de reserva de voos, que conjugam esse serviço, com serviços de reserva de transporte para o aeroporto ou alugueres de automóveis e hotéis. Dada a sua natureza orientada a serviços *web,* normalmente disponíveis através de tradicionais sítios *web,* o seu fator diferenciador reside na experiência de utilização, a qual pode ser uma experiência de utilização superior, quando bem-feita.

Com a realização deste trabalho pode-se concluir que a arquitetura usada pode servir para outro tipo de aplicações, cumprindo-se, assim, um dos objetivos desta dissertação: investigar de que forma a construção de serviços integrados com os sistemas locais atuais pode abrir espaço para novas funcionalidades, sem destruir o *software* legado já existente, o que se comprovou neste trabalho.

6.1 Trabalho futuro

Após a conclusão deste trabalho é importante analisar quais os pontos que podem ser alvo de uma investigação mais aprofundada. Um aspeto a aprofundar é relativo à comparação entre a utilização de serviços que recorrem a SOAP e a WSDL e serviços REST que tipicamente recorrem apenas a http. Seria importante perceber, em detalhe e com situações concretas, a diferença entre os mecanismos oferecidos pelas especificações SOAP e a alternativa oferecida para implementação em serviços REST.

Outro aspeto que seria importante avaliar seria o grau de interoperabilidade dos serviços desenvolvidos com tecnologias diferentes da tecnologia Microsoft .NET. Para um verdadeiro serviço interoperável, as barreiras existentes devem ser reduzidas ao mínimo possível.

Glossário

- **Desktop:** termo relacionado com os computadores pessoais de uso numa única localização. Historicamente os primeiros computadores pessoais ficavam assentes em cima da secretária (do inglês *desk*). O termo é usado para associar também aplicações desenvolvidas para os sistemas operativos desses computadores, daí a utilização da expressão "aplicações de desktop".
- **Endpoint:** ponto de acesso e comunicação com um serviço. Os endpoints providenciam aos clientes as informações necessárias para a comunicação com o serviço.
- **Framework:** conjunto de conceitos comuns, que providencia uma funcionalidade genérica. Uma framework é usada para resolver problemas de um domínio específico, permitindo ao programador focar-se mais nos problemas de negócio e menos com o *plumbing code.*
- **Plumbing Code:** código usado sempre da mesma forma e repetitivamente em tarefas diferentes. Normalmente é código necessário para que o sistema funcione, mas que pouco ou nada tem a ver com o problema de negócio a resolver.
- **Query:** um pedido de informações a um serviço ou a uma base de dados.
- **Toolkit:** bibliotecas que auxiliam o processo de desenvolvimento.
- **Workflow:** sequência de passos de acordo com um conjunto de regras para a automação de um processo de negócio.

Bibliografia

AMRHEIN, D. & QUINT, S. 2009. *Cloud computing for the enterprise: Part 1: Capturing the cloud* [Online]. IBM. Disponível: http://www.ibm.com/developerworks/websphere/techjournal/0904_amrhein/0904_amrhein.html [Data de Acesso 02-03 2013].

ARSANJANI, A. 2004. *Service-oriented modeling and architecture* [Online]. IBM. Disponível: http://www.ibm.com/developerworks/library/ws-soa-design1/ [Data de Acesso 02-03 2013].

BEAUBOUEF, B. 2012. *Fact or Fiction: Hybrid ERP Deployments* [Online]. Disponível: http://gbeaubouef.wordpress.com/2012/04/27/hybrid-erp/ [Data de Acesso 02-03 2013].

BOWEN, F. 2009. *How SOA can ease your move to cloud computing* [Online]. IBM. Disponível: http://www-01.ibm.com/software/solutions/soa/newsletter/nov09/article_soaandcloud.html [Data de Acesso 02-03 2013].

BRUMFIELD, B., COX, G., HILL, D., NOYES, B., PULEIO, M. & SHIFFLETT, K. 2011. *Developer's Guide to Microsoft Prism 4: Building Modular MVVM Applications with Windows Presentation Foundation and Microsoft Silverlight*, Microsoft Press.

ENDREI, M., ANG, J., ARSANJANI, A., CHUA, S., COMTE, P., KROGDAHL, P., LUO, M. & NEWLING, T. 2004. *Patterns : service-oriented architecture and Web services,* United States, IBM, International Technical Support Organization.

ERL, T. 2005. Service-oriented architecture concepts, technology, and design. Upper Saddle River, NJ: Prentice Hall Professional Technical Reference,.

FIELDING, R. T. 2000. *Architectural styles and the design of network-based software architectures.* University of California, Irvine.

FIELDING, R. T. 2008. *REST APIs must be hypertext-driven* [Online]. Disponível: http://roy.gbiv.com/untangled/2008/rest-apis-must-be-hypertext-driven [Data de Acesso 04-15 2013].

FOWLER, M. 2010. *Richardson Maturity Model* [Online]. Disponível: http://martinfowler.com/articles/richardsonMaturityModel.html [Data de Acesso 04-15 2013].

GUDGIN, M. H., MARC; MENDELSOHN NOAH; MOREAU, JEAN-JACQUES; NIELSEN, HENRIK FRYSTYK; KARMARKAR, ANISH; LAFON, YVES;. 2007. *SOAP Version 1.2 Part 1: Messaging Framework (Second Edition)* [Online]. Disponível: http://www.w3.org/TR/soap12-part1/ [Data de Acesso 04-19 2013].

HUHNS, M. N. & SINGH, M. P. 2005. Service-oriented computing: key concepts and principles. *Internet Computing, IEEE,* 9**,** 75-81.

KRILL, P. 2009. *The cloud-SOA connection* [Online]. Info World. Disponível: http://www.infoworld.com/d/cloud-computing/cloud-soa-connection-724 [Data de Acesso 02-03 2013].

LUTHRIA, H. & RABHI, F. A. 2012. Service-Oriented Architectures: Myth or Reality? *Software, IEEE,* 29**,** 46-52.

MARSTON, S., LI, Z., BANDYOPADHYAY, S., ZHANG, J. & GHALSASI, A. 2011. Cloud computing — The business perspective. *Decision Support Systems,* 51**,** 176-189.

MELL, P., GRANCE, T. & INFORMATION TECHNOLOGY LABORATORY (NATIONAL INSTITUTE OF STANDARDS AND TECHNOLOGY). COMPUTER SECURITY DIVISION. 2011. The NIST definition of cloud computing. *NIST special publication 800-145.* Gaithersburg, MD: Computer Security Division, Information Technology Laboratory, National Institute of Standards and Technology,.

MICROSOFT. 2013. *App Architecture* [Online]. Disponível: http://msdn.microsoft.com/library/windows/apps/br211361.aspx [Data de Acesso 24-06-2013.

OASIS 2006. Reference Model for Service Oriented Architecture.

RICHARDSON, L. & RUBY, S. 2007. *Restful web services*, O'Reilly.

ROB HIGH, J., KINDER, S. & GRAHAM, S. 2005. IBM's SOA Foundation. An Architectural Introduction and Overview.

SIRANGI, V. B., VIJAY & BAJPAI, R. 2012. Migration Considerations for Windows 8 Style App.

W3SCHOOLS. *The WSDL Document Structure* [Online]. Disponível: http://www.w3schools.com/WebServices/ws_wsdl_documents.asp [Data de Acesso 06-05 2013].

WEBBER, J., PARASTATIDIS, S. & ROBINSON, I. 2010. *REST in Practice: Hypermedia and Systems Architecture*, O'Reilly Media, Inc.